Аннотация.

Язык “C”(произносится “си”) - это универсальный язык программирования, для которого характерны экономичность выражения, современный поток управления и структуры данных, богатый набор операторов. Язык “C” не является ни языком “очень высокого уровня”, ни “большим” языком, и не предназначается для некоторой специальной области применения. но отсутствие ограничений и общность языка делают его более удобным и эффективным для многих задач, чем языки, предположительно более мощные.

Язык “C”, первоначально предназначавшийся для написания операционной системы “UNIX” на ЭВМ DEC PDP-11, был разработан и реализован на этой системе Деннисом Ричи. Операционная система, компилятор с языка “C” и по существу все прикладные программы системы “UNIX” (включая все программное обеспечение, использованное при подготовке этой книги) написаны на “C”. Коммерческие компиляторы с языка “C” существуют также на некоторых других ЭВМ, включая IBM SYSTEM/370, HONEYWELL 6000, INTERDATA 8/32. Язык “C”, однако, не связан с какими-либо определенными аппаратными средствами или системами, и на нем легко писать программы, которые можно пропускать без изменений на любой ЭВМ, имеющей “C”-компилятор.

Эта книга предназначена для того, чтобы помочь читателю научиться программировать на языке “C”. Она сдержит учебное введение, цель которого - позволить новым пользователям начать программировать как можно быстрее, отдельные главы по всем основным особенностям языка и справочное руководство. Обучение построено в основном на чтении, написании и разборе примеров, а не голой формулировке правил. Примеры, приводимые в книге, по большей части являются законченными реальными программами, а не отдельными фрагментами. Все примеры были проверены непосредственно с текста книги, где они напечатаны в виде, пригодном для ввода в машину. Кроме указаний о том, как сделать использование языка более эффективным, мы также пытались, где это возможно, проиллюстрировать полезные алгоритмы и принципы хорошего стиля и разумной разработки.

Настоящая книга не является вводным курсом в программирование; она предполагает определенное знакомство с основными понятиями программирования такими как переменные, операторы присваивания, циклы, функции. Тем не менее и новичок в программировании должен оказаться в состоянии читать подряд и освоиться с языком, хотя при этом была бы полезной помощь более опытного коллеги.

По нашему опыту , “C” показал себя приятным, выразительным и разносторонним языком на широком множестве разнообразных программ. Его легко выучить , и он не теряет своих качеств с ростом опыта программиста. Мы надеемся , что эта книга поможет вам хорошо его использовать.

Вдумчивая критика и предложения многих наших друзей и коллег очень много добавили как для самой книги, так и для нашего удовольствия при ее написании. В частности, Майк Биапси, Джим Блю, Стью Фельдман, Доуг Мак-Илрой, Билл Рум, Боб Розин и Ларри Рослер тщательно прочитали множество вариантов. Мы также обязаны Элю Ахо, Стиву Борну, Дэву Двораку, Чаку Хэлею, Дебби Хэлей, Мариону Харрису, Рику Холту, Стиву Джонсону, Джону Машею, Бобу Митцу, Ральфу Мьюа, Питеру Нельсону, Эллиоту Пинсону, Биллу Плагеру, Джерри Спиваку, Кену Томпсону и Питеру Вейнбергеру за полезные замечания на различных этапах и Майку Лоску и Джо Осанна за неоценимую помощь при печатании книги. Брайен В. Керниган Деннис М. Ричи

Содержание.

Аннотация

**0.1. Введение**

**1. Учебное введение**

1.1. Начинаем

1.2. Переменные и арифметика

1.3. Оператор FOR

1.4. Символические константы

1.5. Набор полезных программ

1.5.1. Ввод и вывод символов

1.5.2. Копирование файла

1.5.3. Подсчет символов

1.5.4. Подсчет строк

1.5.5. Подсчет слов

1.6. Массивы

1.7. Функции

1.8. Аргументы - вызов по значению

1.9. Массивы символов

1.10. Область действия: внешние переменные

1.11. Резюме

**2. Типы, операции и выражения**

2.1. Имена переменных

2.2. Типы и размеры данных

2.3. Константы

2.3.1. Символьная константа

2.3.2. Константное выражение

2.3.3. Строчная константа

2.4. Описания

2.5. Арифметические операции

2.6. Операции отношения и логические операции

2.7. Преобразование типов

2.8. Операции увеличения и уменьшения

2.9. Побитовые логические операции

2.10. Операции и выражения присваивания

2.11. Условные выражения

2.12. Старшинство и порядок вычисления

**3. Поток управления**

3.1. Операторы и блоки

3.2. IF - ELSE

3.3. ELSE - IF

3.4. Переключатель

3.5. Циклы - WHILE и FOR

3.6. Цикл DO - WHILE

3.7. Оператор BREAK

3.8. Оператор CONTINUE

3.9. Оператор GOTO и метки

**4. Функции и структура программ**

4.1. Основные сведения

4.2. Функции, возвращающие нецелые значения

4.3. Еще об аргументах функций

4.4. Внешние переменные

4.5. Правила, определяющие область действия

4.5.1. Область действия

4.6. Статические переменные

4.7. Регистровые переменные

4.8. Блочная структура

4.9. Инициализация

4.10. Рекурсия

4.11. Препроцессор языка “C”

4.11.1. Включение файлов

4.11.2. Mакроподстановка

**5. Указатели и массивы**

5.1. Указатели и адреса

5.2. Указатели и аргументы функций

5.3. указатели и массивы

5.4. Адресная арифметика

5.5. указатели символов и функции

5.6. Указатели - не целые

5.7. Многомерные массивы

5.8. Массивы указателей; указатели указателей

5.9. Инициализация массивов указателей

5.10. Указатели и многомерные массивы

5.11. Командная строка аргументов

5.12. Указатели на функции

**6. Структуры**

6.1. Основные сведения

6.2. Структуры и функции

6.3. Массивы структур

6.4. Указатели на структуры

6.5. Структуры, ссылающиеся на себя

6.6. Поиск в таблице

6.7. Поля

6.8. Объединения

6.9. Определение типа

**7. Ввод и вывод**

7.1. Обращение к стандартной библиотеке

7.2. Стандартный ввод и вывод - функции GETCHAR и PUTCHAR

7.3. Форматный вывод - функция PRINTF

7.4. Форматный ввод - функция SCANF

7.5. Форматное преобразование в памяти

7.6. Доступ к файлам

7.7. Обработка ошибок - STDERR и EXIT

7.8. Ввод и вывод строк

7.9. Несколько разнообразных функций

7.9.1. Проверка вида символов и преобразования

7.9.2. Функция UNGETC

7.9.3. Обращение к системе

7.9.4. Управление памятью

**8. Интерфейс системы UNIX**

8.1. Дескрипторы файлов

8.2. Низкоуровневый ввод/вывод - операторы READ и WRITE

8.3. Открытие, создание, закрытие и расщепление (UNLINK)

8.4. Произвольный доступ - SEEK и LSEEK

8.5. Пример - реализация функций FOPEN и GETC

8.6. Пример - распечатка справочников

8.7. Пример - распределитель памяти

**9. Приложение а: справочное руководство по языку 'C'.**

9.1. Введение

**10. Лексические соглашения**

10.1. Комментарии

10.2. Идентификаторы (имена)

10.3. Ключевые слова

10.4. Константы

10.4.1. Целые константы

10.4.2. Явные длинные константы

10.4.3. Символьные константы

10.4.4. Плавающие константы

10.5. Строки

10.6. Характеристики аппаратных средств

**11. Синтаксическая нотация**

**12. Что в имени тебе моем?**

**13. Объекты и L-значения**

**14. Преобразования**

14.1. Символы и целые

14.2. Типы FLOAT и DOUBLE

14.3. Плавающие и целочисленные величины

14.4. Указатели и целые

14.5. Целое без знака

14.6. Арифметические преобразования

**15. Выражения**

15.1. Первичные выражения

15.2. Унарные операции

15.3. Мультипликативные операции

15.4. Аддитивные операции

15.5. Операции сдвига

15.6. Операции отношения

15.7. Операции равенства

15.8. Побитовая операция 'и'

15.9. Побитовая операция исключающего 'или'

15.10. Побитовая операция включающего 'или'

15.11. Логическая операция 'и'

15.12. Операция логического 'или'

15.13. Условная операция

15.14. Операция присваивания

15.15. Операция запятая

**16. Описания**

16.1. Спецификаторы класса памяти

16.2. Спецификаторы типа

16.3. Описатели

16.4. Смысл описателей

16.5. Описание структур и объединений

16.6. Инициализация

16.7. Имена типов

16.8. TYPEDEF

**17. Операторы**

17.1. Операторное выражение

17.2. Составной оператор (или блок)

17.3. Условные операторы

17.4. Оператор WHILE

17.5. Оператор DO

17.6. Оператор FOR

17.7. Оператор SWITCH

17.8. Оператор BREAK

17.9. Оператор CONTINUE

17.10. Оператор возврата

17.11. Оператор GOTO

17.12. Помеченный оператор

17.13. Пустой оператор

**18. Внешние определения**

18.1. Внешнее определение функции

18.2. Внешние определения данных

**19. Правила, определяющие область действия**

19.1. Лексическая область действия

19.2. Область действия внешних идентификаторов

**20. Строки управления компилятором**

20.1. Замена лексем

20.2. Включение файлов

20.3. Условная компиляция

**21. Неявные описания**

**22. Снова о типах**

22.1. Структуры и объединения

22.2. Функции

22.3. Массивы, указатели и индексация

22.4. Явные преобразования указателей

**23. Константные выражения**

**24. Соображения о переносимости**

**25. Анахронизмы**

**26. Сводка синтаксических правил**

26.1. Выражения

26.2. Описания

26.3. Операторы

26.4. Внешние определения

26.5. Препроцессор

**27. Присваивание структуры**

**28. Тип перечисления**

**29. Таблица изображений непечатных символов языка “C”.**

0.1. Введение.

Язык “C” является универсальным языком программирования. Он тесно связан с операционной системой “UNIX” , так как был развит на этой системе и так как “UNIX” и ее программное обеспечение написано на “C”. Сам язык , однако, не

связан с какой-либо одной операционной системой или машиной;

и хотя его называют языком системного программирования, так

как он удобен для написания операционных систем, он с равным

успехом использовался при написании больших вычислительных

программ, программ для обработки текстов и баз данных.

Язык “C” - это язык относительно “низкого уровня”. В

такой характеристике нет ничего оскорбительного; это просто

означает, что “C” имеет дело с объектами того же вида, что и

большинство ЭВМ, а именно, с символами, числами и адресами.

Они могут объединяться и пересылаться посредством обычных

арифметических и логических операций, осуществляемых реаль-

ными ЭВМ.

В языке “C” отсутствуют операции, имеющие дело непос-

редственно с составными объектами, такими как строки симво-

лов, множества, списки или с массивами, рассматриваемыми как

целое. Здесь, например, нет никакого аналога операциям PL/1,

оперирующим с целыми массивами и строками. Язык не предос-

тавляет никаких других возможностей распределения памяти,

кроме статического определения и механизма стеков, обеспечи-

ваемого локальными переменных функций; здесь нет ни

“куч”(HEAP), ни “сборки мусора”, как это предусматривается в

АЛГОЛЕ-68. Наконец, сам по себе “C” не обеспечивает никаких

возможностей ввода-вывода: здесь нет операторов READ или

WRITE и никаких встроенных методов доступа к файлам. Все эти

механизмы высокого уровня должны обеспечиваться явно вызыва-

емыми функциями.

Аналогично, язык “C” предлагает только простые, после-

довательные конструкции потоков управления: проверки, циклы,

группирование и подпрограммы, но не мультипрограммирование,

параллельные операции, синхронизацию или сопрограммы.

Хотя отсутствие некоторых из этих средств может выгля-

деть как удручающая неполноценность (“выходит, что я должен

обращаться к функции, чтобы сравнить две строки символов

?!”), но удержание языка в скромных размерах дает реальные

преимущества. Так как “C” относительно мал, он не требует

много места для своего описания и может быть быстро выучен.

Компилятор с “C” может быть простым и компактным. Кроме то-

го, компиляторы легко пишутся; при использовании современной

технологии можно ожидать написания компилятора для новой ЭВМ

за пару месяцев и при этом окажется, что 80 процентов прог-

раммы нового компилятора будет общей с программой для уже

существующих компиляторов. Это обеспечивает высокую степень

мобильности языка. Поскольку типы данных и стуктуры управле-

ния, имеющиеся в “C”, непосредственно поддерживаются боль-

шинством существующих ЭВМ, библиотека, необходимая во время

прогона изолированных программ, оказывается очень маленькой.

На PDP -11, например, она содержит только программы для

32-битового умножения и деления и для выполнения программ

ввода и вывода последовательностей. Конечно, каждая реализа-

ция обеспечивает исчерпывающую, совместимую библиотеку функ-

ций для выполнения операций ввода-вывода, обработки строк и

распределения памяти, но так как обращение к ним осуществля-

ется только явно, можно , если необходимо, избежать их вызо-

ва; эти функции могут быть компактно написаны на самом “C”.

* 8 -

Опять же из-за того , что язык “C” отражает возможности

современных компьютеров, программы на “C” оказываются доста-

точно эффективными, так что не возникает побуждения писать

вместо этого программы на языке ассемблера. Наиболее убеди-

тельным примером этого является сама операционная система

“UNIX”, которая почти полностью написана на “C”. Из 13000

строк программы системы только около 800 строк самого низко-

го уровня написаны на ассемблере. Кроме того, по существу

все прикладное программное обеспечение системы “UNIX” напи-

сано на “C”; подавляющее большинство пользователей системы

“UNIX”(включая одного из авторов этой книги) даже не знает

языка ассемблера PDP-11.

Хотя “C” соответствует возможностям многих ЭВМ, он не

зависит от какой-либо конкретной архитектуры машины и в силу

этого без особых усилий позволяет писать “переносимые” прог-

раммы, т.е. программы, которые можно пропускать без измене-

ний на различных аппаратных средствах. В наших кругах стал

уже традицией перенос программного обеспечения, разработан-

ного на системе “UNIX”, на системы ЭВМ: HONEYWELL, IBM и

INTERDATA. Фактически компиляторы с “C” и программное обес-

печение во время прогона программ на этих четырех системах,

по-видимому, гораздо более совместимы, чем стандартные вер-

сии фортрана американского национального института стандар-

тов (ANSI). Сама операционная система “UNIX” теперь работает

как на PDP-11, так и на INTERDATA 8/32. За исключением прог-

рамм, которые неизбежно оказываются в некоторой степени ма-

шинно-зависимыми, таких как компилятор, ассемблер и отлад-

чик. Написанное на языке “C” программное обеспечение иден-

тично на обеих машинах. Внутри самой операционной системы

7000 строк программы, исключая математическое обеспечение

языка ассемблера ЭВМ и управления операциями ввода-вывода,

совпадают на 95 процентов.

Программистам, знакомым с другими языками, для сравне-

ния и противопоставления может оказаться полезным упоминание

нескольких исторических, технических и философских аспектов

“C”.

Многие из наиболее важных идей “C” происходят от гораз-

до более старого, но все еще вполне жизненного языка BCPL ,

разработанного Мартином Ричардсом. Косвенно язык BCPL оказал

влияние на “C” через язык “B”, написанный Кеном Томпсоном в

1970 году для первой операционной системы “UNIX” на ЭВМ

PDP-7.

Хотя язык “C” имеет несколько общих с BCPL характерных

особенностей, он никоим образом не является диалектом пос-

леднего. И BCPL и “B” - “безтипные” языки; единственным ви-

дом данных для них являются машинное слово, а доступ к дру-

гим объектам реализуется специальными операторами или обра-

щением к функциям. В языке “C” объектами основных типов дан-

ных являются символы, целые числа нескольких размеров и чис-

ла с плавающей точкой. Кроме того, имеется иерархия произ-

водных типов данных, создаваемых указателями, массивами,

структурами, объединениями и функциями.

* 9 -

Язык “C” включает основные конструкции потока управле-

ния, требуемые для хорошо структуированных программ: группи-

рование операторов, принятие решений (IF), циклы с проверкой

завершения в начале (WHILE, FOR) или в конце (DO) и выбор

одного из множества возможных вариантов (SWITCH). (Все эти

возможности обеспечивались и в BCPL, хотя и при несколько

отличном синтаксисе; этот язык предчувствовал наступившую

через несколько лет моду на структурное программирование).

В языке “C” имеются указатели и возможность адресной

арифметики. Аргументы передаются функциям посредством копи-

рования значения аргумента , и вызванная функция не может

изменить фактический аргумент в вызывающей программе. Если

желательно добиться “вызова по ссылке”, можно неявно пере-

дать указатель, и функция сможет изменить объект, на который

этот указатель указывает. Имена массивов передаются указани-

ем начала массивов, так что аргументы типа массивов эффек-

тивно вызываются по ссылке.

К любой функции можно обращаться рекурсивно, и ее ло-

кальные переменные обычно “автоматические”, т.е. Создаются

заново при каждом обращении. Описание одной функции не может

содержаться внутри другой, но переменные могут описываться в

соответствии с обычной блочной структурой. Функции в “C” -

программе могут транслироваться отдельно. переменные по от-

ношению к функции могут быть внутренними, внешними, но из-

вестными только в пределах одного исходного файла, или пол-

ностью глобальными. Внутренние переменные могут быть автома-

тическими или статическими. Автоматические переменные для

большей эффективности можно помещать в регистры, но объявле-

ние регистра является только указанием для компилятора и ни-

как не связано с конкретными машинными регистрами.

Язык “C” не является языком со строгими типами в смысле

паскаля или алгола 68. Он сравнительно снисходителен к пре-

образованию данных, хотя и не будет автоматически преобразо-

вывать типы данных с буйной непринужденностью языка PL/1.

Существующие компиляторы не предусматривают никакой проверки

во время выполнения программы индексов массивов, типов аргу-

ментов и т.д.

В тех ситуациях, когда желательна строгая проверка ти-

пов, используется специальная версия компилятора. Эта прог-

рамма называется LINT очевидно потому, она выбирает кусочки

пуха из вашей программы. Программа LINT не генерирует машин-

ного кода, а делает очень строгую проверку всех тех сторон

программы, которые можно проконтролировать во время компиля-

ции и загрузки. Она определяет несоответствие типов, несов-

местимость аргументов, неиспользованные или очевидным обра-

зом неинициализированные переменные, потенциальные трудности

переносимости и т.д. Для программ,которые благополучно про-

ходят через LINT, гарантируется отсутствие ошибок типа при-

мерно с той же полнотой, как и для программ, написанных,

например, на АЛГОЛЕ-68. Другие возможности программы LINT

будут отмечены, когда представится соответствующий случай.

* 10 -

Наконец, язык “C”, подобно любому другому языку, имеет

свои недостатки. Некоторые операции имеют неудачное старшин-

ство; некоторые разделы синтаксиса могли бы быть лучше; су-

шествует несколько версий языка, отличающихся небольшими де-

талями. Тем не менее язык “C” зарекомендовал себя как исклю-

чительно эффективный и выразительный язык для широкого раз-

нообразия применений программирования.

Содержание книги организовано следующим образом. Глава

1 является учебным введением в центральную часть языка “C”.

Цель - позволить читателю стартовать так быстро,как только

возможно, так как мы твердо убеждены, что единственный спо-

соб изучить новый язык - писать на нем программы. При этом ,

однако, предполагается рабочее владение основными элементами

программирования; здесь не объясняется, что такое ЭВМ или

компилятор, не поясняется смысл выражений типа N=N+1. Хотя

мы и пытались, где это возможно, продемонстрировать полезную

технику программирования. Эта книга не предназначается быть

справочным руководством по структурам данных и алгоритмам;

там, где мы вынуждены были сделать выбор, мы концентрирова-

лись на языке.

В главах со 2-й по 6-ю различные аспекты “C” излагаются

более детально и несколько более формально, чем в главе 1,

хотя ударение по-прежнему делается на разборе примеров за-

конченных, полезных программ, а не на отдельных фрагментах.

В главе 2 обсуждаются основные типы данных, операторы и

выражения. В главе 3 рассматриваются управляющие операторы:

IF-ELSE ,WHILE ,FOR и т.д. Глава 4 охватывает функции и

структуру программы - внешние переменные, правила определен-

ных областей действия описания и т.д. В главе 5 обсуждаются

указатели и адресная арифметика. Глава 6 содержит подробное

описание структур и объединений.

В главе 7 описывается стандартная библиотека ввода-вы-

вода языка “C”, которая обеспечивает стандартный интерфейс с

операционной системой. Эта библиотека ввода-вывода поддержи-

вается на всех машинах, на которых реализован “C”, так что

программы, использующие ее для ввода, вывода и других сис-

темных функций, могут переноситься с одной системы на другую

по существу без изменений.

В главе 8 описывается интерфейс между “C” - программами

и операционной системой “UNIX”. Упор делается на ввод-вывод,

систему файлов и переносимость. Хотя некоторые части этой

главы специфичны для операционной системы “UNIX”, програм-

мисты, не использующие “UNIX”, все же должны найти здесь по-

лезный материал, в том числе некоторое представление о том,

как реализована одна версия стандартной библиотеки и предло-

жения для достижения переносимости программы.

Приложение A содержит справочное руководство по языку

“C”. Оно является “официальным” изложением синтаксиса и се-

мантики “C” и (исключая чей-либо собственный компилятор)

окончательным арбитром для всех двусмысленностей и упущений

в предыдущих главах.

* 11 -

Так как “C” является развивающимся языком, реализован-

ным на множестве систем, часть материла настоящей книги мо-

жет не соответствовать текущему состоянию разработки на ка-

кой-то конкретной системе. Мы старались избегать таких проб-

лем и предостерегать о возможных трудностях. В сомнительных

случаях, однако, мы обычно предпочитали описывать ситуацию

для системы “UNIX” PDP-11 , так как она является средой для

большинства программирующих на языке “C”. В приложении а

также описаны расхождения в реализациях языка “C” на основ-

ных системах.

1. Учебное введение.

Давайте начнем с быстрого введения в язык “C”. Наша

цель - продемонстрировать существенные элементы языка на ре-

альных программах, не увязая при этом в деталях, формальных

правилах и исключениях. В этой главе мы не пытаемся изложить

язык полностью или хотя бы строго (разумеется, приводимые

примеры будут корректными). Мы хотим как можно скорее довес-

ти вас до такого уровня, на котором вы были бы в состоянии

писать полезные программы, и чтобы добиться этого, мы сосре-

дотачиваемся на основном: переменных и константах, арифмети-

ке, операторах передачи управления, функциях и элементарных

сведениях о вводе и выводе. Мы совершенно намеренно оставля-

ем за пределами этой главы многие элементы языка “C”, кото-

рые имеют первостепенное значение при написании больших

программ, в том числе указатели, сртуктуры, большую часть из

богатого набора операторов языка “C”, несколько операторов

передачи управления и несметное количество деталей.

Такой подход имеет, конечно, свои недостатки. Самым су-

щественным является то, что полное описание любого конкрет-

ного элемента языка не излагается в одном месте, а поясне-

ния, в силу краткости, могут привести к неправильному истол-

кованию. Кроме того, из-за невозможности использовать всю

мощь языка, примеры оказываются не столь краткими и элегант-

ными, как они могли бы быть. И хотя мы старались свести эти

недостатки к минимуму, все же имейте их ввиду.

Другой недостаток состоит в том, что последующие главы

будут неизбежно повторять некоторые части этой главы. Мы на-

деемся, что такое повторение будет скорее помогать, чем раз-

дражать.

Во всяком случае, опытные программисты должны оказаться

в состоянии проэкстраполировать материал данной главы на

свои собственные программистские нужды. Начинающие же должны

в дополнение писать аналогичные маленькие самостоятельные

программы. И те, и другие могут использовать эту главу как

каркас, на который будут навешиваться более подробные описа-

ния, начинающиеся с главы 2.

1.1. Hачинаем. Единственный способ освоить новый язык

программирования - писать на нем программы. Первая програм-

ма, которая должна быть написана, - одна для всех языков:

напечатать слова : HELLO, WORLD.

Это - самый существенный барьер; чтобы преодолеть его,

вы должны суметь завести где-то текст программы, успешно его

скомпилировать, загрузить, прогнать и найти, где оказалась

ваша выдача. Если вы научились справляться с этими техничес-

кими деталями, все остальное сравнительно просто.

* 12 -

Программа печати “HELLO, WORLD” на языке “C” имеет вид:

MAIN ()

{

PRINTF(“HELLO, WORLD\N”);

}

Как пропустить эту программу - зависит от используемой

вами системы. В частности, на операционной системе “UNIX” вы

должны завести исходную программу в файле, имя которого

оканчивается на “.C” , например, HELLO.C , и затем скомпили-

ровать ее по команде

CC HELLO.C

Если вы не допустили какой-либо небрежности , такой как

пропуск символа или неправильное написание, компиляция прой-

дет без сообщений и будет создан исполняемый файл с именем

а.OUT . Прогон его по команде

A.OUT

приведет к выводу

HELLO, WORLD

На других системах эти правила будут иными; проконсуль-тируйтесь с местным авторитетом.

Упражнение 1-1.

Пропустите эту программу на вашей системе. Попробуйте

не включать различные части программы и посмотрите какие со-

общения об ошибках вы при этом получите.

Теперь некоторые пояснения к самой программе. Любая

“C”-программа, каков бы ни был ее размер, состоит из одной

или более “функций”, указывающих фактические операции

компьютера, которые должны быть выполнены. Функции в языке

“C” подобны функциям и подпрограммам фортрана и процедурам

PL/1, паскаля и т.д. В нашем примере такой функцией является

MAIN. Обычно вы можете давать функциям любые имена по вашему

усмотрению, но MAIN - это особое имя; выполнение вашей прог-

раммы начинается сначала с функции MAIN. Это означает, что

каждая программа должна в каком-то месте содержать функцию с

именем MAIN. Для выполнения определенных действий функция

MAIN обычно обращается к другим функциям, часть из которых

находится в той же самой программе, а часть - в библиотеках,

содержащих ранее написанные функции.

* 13 -

Одним способом обмена данными между функциями является

передача посредством аргументов. Круглые скобки, следующие

за именем функции, заключают в себе список аргументов; здесь

маIN - функция без аргументов, что указывается как (). Опе-

раторы, составляющие функцию, заключаются в фигурные скобки

{ и }, которые аналогичны DO-END в PL/1 или BEGIN-END в ал-

голе, паскале и т.д. Обращение к функции осуществляется ука-

занием ее имени, за которым следует заключенный в круглые

скобки список аргументов. здесь нет никаких операторов CALL,

как в фортране или PL/1. Круглые скобки должны присутство-

вать и в том случае, когда функция не имеет аргументов.

Строка

PRINTF(“HELLO, WORLD\N”);

является обращением к функции, которое вызывает функцию

с именем PRINTF и аргуметом “HELLO, WORLD\N”. Функция PRINTF

является библиотечной функцией, которая выдает выходные дан-

ные на терминал (если только не указано какое-то другое мес-

то назначения). В данном случае печатается строка символов,

являющаяся аргументом функции.

Последовательность из любого количества символов, зак-

люченных в удвоенные кавычки “...”, называется 'символьной

строкой' или 'строчной константой'. Пока мы будем использо-

вать символьные строки только в качестве аргументов для

PRINTF и других функций.

Последовательность \N в приведенной строке является

обозначением на языке “C” для 'символа новой строки', кото-

рый служит указанием для перехода на терминале к левому краю

следующей строки. Если вы не включите \N (полезный экспери-

мент), то обнаружите, что ваша выдача не закончится перехо-

дом терминала на новую строку. Использование последователь-

ности \N - единственный способ введения символа новой строки

в аргумент функции PRINTF; если вы попробуете что-нибудь

вроде

PRINTF(“HELLO, WORLD

“);

то “C”-компилятор будет печатать злорадные диагностические

сообщения о недостающих кавычках.

Функция PRINTF не обеспечивает автоматического перехода

на новую строку, так что многократное обращение к ней можно

использовать для поэтапной сборки выходной строки. Наша пер-

вая программа, печатающая идентичную выдачу, с точно таким

же успехом могла бы быть написана в виде

MAIN()

{

PRINTF(“HELLO, “);

PRINTF(“WORLD”);

PRINTF(“\N”);

}

* 14 -

Подчеркнем, что \N представляет только один символ. Ус-

ловные 'последовательности', подобные \N , дают общий и до-

пускающий расширение механизм для представления трудных для

печати или невидимых символов. Среди прочих символов в языке

“C” предусмотрены следующие: \т - для табуляции, \B - для

возврата на одну позицию, \” - для двойной кавычки и \\ для

самой обратной косой черты.

Упражнение 1-2.

Проведите эксперименты для того, чтобы узнать что прои-

зойдет, если в строке, являющейся аргументом функции PRINTF

будет содержаться \X, где X - некоторый символ, не входящий

в вышеприведенный список.

1.2. Переменные и арифметика.

Следующая программа печатает приведенную ниже таблицу

температур по Фаренгейту и их эквивалентов по стоградусной

шкале Цельсия, используя для перевода формулу

C = (5/9)\*(F-32).

0 -17.8

20 -6.7

40 4.4

60 15.6

... ...

260 126.7

280 137.8

300 140.9

Теперь сама программа:

/\* PRINT FAHRENHEIT-CELSIUS TABLE

FOR F = 0, 20, ..., 300 \*/

MAIN()

{

INT LOWER, UPPER, STEP;

FLOAT FAHR, CELSIUS;

LOWER = 0; /\* LOWER LIMIT OF TEMPERATURE

TABLE \*/

UPPER =300; /\* UPPER LIMIT \*/

STEP = 20; /\* STEP SIZE \*/

FAHR = LOWER;

WHILE (FAHR <= UPPER) {

CELSIUS = (5.0/9.0) \* (FAHR -32.0);

PRINTF(“%4.0F %6.1F\N”, FAHR, CELSIUS);

FAHR = FAHR + STEP;

}

}

* 15 -

Первые две строки

/\* PRINT FAHRENHEIT-CELSIUS TABLE

FOR F = 0, 20, ..., 300 \*/

являются комментарием, который в данном случае кратко пояс-

няет, что делает программа. Любые символы между /\* и \*/ иг-

норируются компилятором; можно свободно пользоваться коммен-

тариями для облегчения понимания программы. Комментарии мо-

гут появляться в любом месте, где возможен пробел или пере-

ход на новую строку.

В языке “C” все переменные должны быть описаны до их ис-

пользования, обычно это делается в начале функции до первого

выполняемого оператора. Если вы забудете вставить описание,

то получите диагностическое сообщение от компилятора. Описа-

ние состоит из типа и списка переменных, имеющих этот тип,

как в

INT LOWER, UPPER, STEP;

FLOAT FAHR, CELSIUS;

Тип INT означает, что все переменные списка целые; тип

FLOAT предназначен для чисел с плавающей точкой, т.е. для

чисел, которые могут иметь дробную часть. Точность как INT ,

TAK и FLOAT зависит от конкретной машины, на которой вы ра-

ботаете. На PDP-11, например, тип INT соответствует 16-бито-

вому числу со знаком, т.е. числу, лежащему между -32768 и

+32767. Число типа FLOAT - это 32-битовое число, имеющее

около семи значащих цифр и лежащее в диапазоне от 10е-38 до

10е+38. В главе 2 приводится список размеров для других ма-

шин.

В языке “C” предусмотрено несколько других основных ти-

пов данных, кроме INT и FLOAT:

CHAR символ - один байт

SHORT короткое целое

LONG длинное целое

DOUBLE плавающее с двойной точностью

Размеры этих объектов тоже машинно-независимы; детали

приведены в главе 2. Имеются также массивы, структуры и об-

ъединения этих основных типов, указатели на них и функ-

ции,которые их возвращают; со всеми ними мы встретимся в

свое время.

Фактически вычисления в программе перевода температур

начинаются с операторов присваивания

LOWER = 0;

UPPER =300;

STEP = 20;

FAHR =LOWER;

которые придают переменным их начальные значения. каждый от-

дельный оператор заканчивается точкой с запятой.

Каждая строка таблицы вычисляется одинаковым образом,

так что мы используем цикл, повторяющийся один раз на стро-

ку. В этом назначение оператора WHILE:

WHILE (FAHR <= UPPER) {

....

}

* 16 -

проверяется условие в круглых скобках. Если оно истинно

(FAHR меньше или равно UPPER), то выполняется тело цикла

(все операторы, заключенные в фигурные скобки { и } ). Затем

вновь проверяется это условие и, если оно истинно, опять вы-

полняется тело цикла. Если же условие не выполняется ( FAHR

превосходит UPPER ), цикл заканчивается и происходит переход

к выполнению оператора, следующего за оператором цикла. Так

как в настоящей программе нет никаких последующих операто-

ров, то выполнение программы завершается.

Тело оператора WHILE может состоять из одного или более

операторов, заключенных в фигурные скобки, как в программе

перевода температур, или из одного оператора без скобок,

как, например, в

WHILE (I < J)

I = 2 \* I;

В обоих случаях операторы, управляемые оператором WHILE,

сдвинуты на одну табуляцию, чтобы вы могли с первого взгляда

видеть, какие операторы находятся внутри цикла. Такой сдвиг

подчеркивает логическую структуру программы. Хотя в языке

“C” допускается совершенно произвольное расположение опера-

торов в строке, подходящий сдвиг и использование пробелов

значительно облегчают чтение программ. Мы рекомендуем писать

только один оператор на строке и (обычно) оставлять пробелы

вокруг операторов. Расположение фигурных скобок менее сущес-

твенно; мы выбрали один из нескольких популярных стилей. Вы-

берите подходящий для вас стиль и затем используйте его пос-

ледовательно.

Основная часть работы выполняется в теле цикла. Темпера-

тура по Цельсию вычисляется и присваивается переменной

CELAIUS оператором

CELSIUS = (5.0/9.0) \* (FAHR-32.0);

причина использования выражения 5.0/9.0 вместо выглядящего

проще 5/9 заключается в том, что в языке “C”, как и во мно-

гих других языках, при делении целых происходит усечение,

состоящее в отбрасывании дробной части результата. Таким об-

разом, результат операции 5/9 равен нулю, и, конечно, в этом

случае все температуры оказались бы равными нулю. Десятичная

точка в константе указывает, что она имеет тип с плавающей

точкой, так что, как мы и хотели, 5.0/9.0 равно 0.5555... .

Мы также писали 32.0 вместо 32 , несмотря на то, что так

как переменная FAHR имеет тип FLOAT , целое 32 автоматически

бы преобразовалось к типу FLOAT ( в 32.0) перед вычитанием.

С точки зрения стиля разумно писать плавающие константы с

явной десятичной точкой даже тогда, когда они имеют целые

значения; это подчеркивает их плавающую природу для просмат-

ривающего программу и обеспечивает то, что компилятор будет

смотреть на вещи так же, как и Вы.

* 17 -

Подробные правила о том, в каком случае целые преобразу-

ются к типу с плаваюшей точкой, приведены в главе 2. Сейчас

же отметим, что присваивание

FAHR = LOWER;

проверка

WHILE (FAHR <= UPPER)

работают, как ожидается, - перед выполнением операций целые

преобразуются в плавающую форму.

Этот же пример сообщает чуть больше о том, как работает

PRINTF. Функция PRINTF фактически является универсальной

функцией форматных преобразований, которая будет полностью

описана в главе 7. Ее первым аргументом является строка сим-

волов, которая должна быть напечатана, причем каждый знак %

указывает, куда должен подставляться каждый из остальных ар-

гументов /второй, третий, .../ и в какой форме он должен пе-

чататься. Например, в операторе

PRINTF(“%4.0F %6.1F\N”, FAHR, CELSIUS);

спецификация преобразования %4.0F говорит, что число с пла-

вающей точкой должно быть напечатано в поле шириной по край-

ней мере в четыре символа без цифр после десятичной точки.

спецификация %6.1F описывает другое число, которое должно

занимать по крайней мере шесть позиций с одной цифрой после

десятичной точки, аналогично спецификациям F6.1 в фортране

или F(6,1) в PL/1. Различные части спецификации могут быть

опущены: спецификация %6F говорит, что число будет шириной

по крайней мере в шесть символов; спецификация %2 требует

двух позиций после десятичной точки, но ширина при этом не

ограничивается; спецификация %F говорит только о том, что

нужно напечатать число с плавающей точкой. Функция PRINTF

также распознает следующие спецификации: %D - для десятично-

го целого, %о - для восьмеричного числа, %х - для шестнадца-

тиричного, %с - для символа, %S - для символьной строки и %%

* для самого символа %.

Каждая конструкция с символом % в первом аргументе функ-

ции PRINTF сочетается с соответствующим вторым, третьим, и

т.д. Аргументами; они должны согласовываться по числу и ти-

пу; в противном случае вы получите бессмысленные результаты.

Между прочим, функция PRINTF не является частью языка

“C”; в самом языке “C” не определены операции ввода-вывода.

Нет ничего таинственного и в функции PRINTF ; это - просто

полезная функция, являющаяся частью стандартной библиотеки

подпрограмм, которая обычно доступна “C”-программам. Чтобы

сосредоточиться на самом языке, мы не будем подробно оста-

навливаться на операциях ввода-вывода до главы 7. В частнос-

ти, мы до тех пор отложим форматный ввод. Если вам надо

ввести числа - прочитайте описание функции SCANF в главе 7,

раздел 7.4. Функция SCANF во многом сходна с PRINTF , но она

считывает входные данные, а не печатает выходные.

* 18 -

Упражнение 1-3.

Преобразуйте программу перевода температур таким обра-

зом, чтобы она печатала заголовок к таблице.

Упражнение 1-4.

Напишите программы печати соответствующей таблицы пере-

хода от градусов цельсия к градусам фаренгейта.

1.3. Оператор FOR.

Как и можно было ожидать, имеется множество различных

способов написания каждой программы. Давайте рассмотрим та-

кой вариант программы перевода температур:

MAIN() /\* FAHRENHEIT-CELSIUS TABLE \*/

{

INT FAHR;

FOR (FAHR = 0; FAHR <= 300; FAHR = FAHR + 20)

PRINTF(“%4D %6.1F\N”, FAHR, (5.0/9.0)\*(FAHR-32.0));

}

Эта программа выдает те же самые результаты, но выглядит

безусловно по-другому. Главное изменение - исключение боль-

шинства переменных; осталась только переменная FAHR , причем

типа INT (это сделано для того, чтобы продемонстрировать

преобразование %D в функции PRINTF). Нижняя и верхняя грани-

цы и размер щага появляются только как константы в операторе

FOR , который сам является новой конструкцией, а выражение,

вычисляющее температуру по цельсию, входит теперь в виде

третьего аргумента функции PRINTF , а не в виде отдельного

оператора присваивания.

Последнее изменение является примером вполне общего пра-

вила языка “C” - в любом контексте, в котором допускается

использование значения переменной некоторого типа, вы можете

использовать выражение этого типа. Так как третий аргумент

функции PRINTF должен иметь значение с плавающей точкой,

чтобы соответствовать спецификации %6.1F, то в этом месте

может встретиться любое выражение плавающего типа.

Сам оператор FOR - это оператор цикла, обобщающий опера-

тор WHILE. Его функционирование должно стать ясным, если вы

сравните его с ранее описанным оператором WHILE . Оператор

FOR содержит три части, разделяемые точкой с запятой. Первая

часть

FAHR = 0

выполняется один раз перед входом в сам цикл. Вторая часть -

проверка, или условие, которое управляет циклом:

FAHR <= 300

это условие проверяется и, если оно истинно, то выполняется

тело цикла (в данном случае только функция PRINTF ). Затем

выполняется шаг реинициализации

* 19 -

FAHR =FAHR + 20

и условие проверяется снова. цикл завершается, когда это ус-

ловие становится ложным. Так же, как и в случае оператора

WHILE , тело цикла может состоять из одного оператора или из

группы операторов, заключенных в фигурные скобки. Инициали-

зирующая и реинициализирующая части могут быть любыми от-

дельными выражениями.

Выбор между операторами WHILE и FOR произволен и основы-

вается на том , что выглядит яснее. Оператор FOR обычно удо-

бен для циклов, в которых инициализация и реинициализация

логически связаны и каждая задается одним оператором, так

как в этом случае запись более компактна, чем при использо-

вании оператора WHILE , а операторы управления циклом сосре-

дотачиваются вместе в одном месте.

Упражнение 1-5.

Модифицируйте программу перевода температур таким обра-

зом, чтобы она печатала таблицу в обратном порядке, т.е. От

300 градусов до 0.

1.4. Символические константы.

Последнее замечание, прежде чем мы навсегда оставим

программу перевода температур. Прятать “магические числа”,

такие как 300 и 20, внутрь программы - это неудачная практи-

ка; они дают мало информации тем, кто, возможно, должен бу-

дет разбираться в этой программе позднее, и их трудно изме-

нять систематическим образом. К счастью в языке “C” предус-

мотрен способ, позволяющий избежать таких “магических чи-

сел”. Используя конструкцию #DEFINE , вы можете в начале

программы определить символическое имя или символическую

константу, которая будет конкретной строкой символов. Впос-

ледствии компилятор заменит все не заключенные в кавычки по-

явления этого имени на соответствующую строку. Фактически

это имя может быть заменено абсолютно произвольным текстом,

не обязательно цифрами

#DEFINE LOWER 0/\* LOWER LIMIT OF TABLE \*/

#DEFINE UPPER 300 /\* UPPER LIMIT \*/

#DEFINE STEP 20 /\* STEP SIZE \*/

MAIN () /\* FAHRENHEIT-CELSIUS TABLE \*/

{

INT FAHR;

FOR (FAHR =LOWER; FAHR <= UPPER; FAHR =FAHR + STEP)

PRINTF(“%4D %6.1F\N”, FAHR, (5.0/9.0)\*(FAHR-32));

}

величины LOWER, UPPER и STEP являются константами и поэ-

тому они не указываются в описаниях. Символические имена

обычно пишут прописными буквами, чтобы их было легко отли-

чить от написанных строчными буквами имен переменных. отме-

тим, что в конце определения не ставится точка с запятой.

Так как подставляется вся строка, следующая за определенным

именем, то это привело бы к слишком большому числу точек с

запятой в операторе FOR .

* 20 -

1.5. Набор полезных программ.

Теперь мы собираемся рассмотреть семейство родственных

программ, предназначенных для выполнения простых операций

над символьными данными. В дальнейшем вы обнаружите, что

многие программы являются просто расширенными версиями тех

прототипов, которые мы здесь обсуждаем.

1.5.1. Ввод и вывод символов.

Стандартная библиотека включает функции для чтения и за-

писи по одному символу за один раз. функция GETCHAR() извле-

кает следующий вводимый символ каждый раз, как к ней обраща-

ются, и возвращает этот символ в качестве своего значения.

Это значит, что после

C = GETCHAR()

переменная 'C' содержит следующий символ из входных данных.

Символы обычно поступают с терминала, но это не должно нас

касаться до главы 7.

Функция PUTCHAR© является дополнением к GETCHAR : в

результате обращения

PUTCHAR ©

содержимое переменной 'C' выдается на некоторый выходной но-

ситель, обычно опять на терминал. Обращение к функциям

PUTCHAR и PRINTF могут перемежаться; выдача будет появляться

в том порядке, в котором происходят обращения.

Как и функция PRINTF , функции GETCHAR и PUTCHAR не со-

держат ничего экстраординарного. Они не входят в состав язы-

ка “C”, но к ним всегда можно обратиться.

1.5.2. Копирование файла.

Имея в своем распоряжении только функции GETCHAR и

PUTCHAR вы можете, не зная ничего более об операциях вво-

да-вывода, написать удивительное количество полезных прог-

рамм. Простейшим примером может служить программа посимволь-

ного копирования вводного файла в выводной. Общая схема име-

ет вид:

ввести символ

WHILE (символ не является признаком конца файла)

вывести только что прочитанный символ

ввести новый символ

программа, написанная на языке “C”, выглядит следующим обра-

зом:

MAIN() /\* COPY INPUT TO OUTPUT; 1ST VERSION \*/

{

INT C;

C = GETCHAR();

WHILE (C != EOF) {

PUTCHAR ©;

C = GETCHAR();

}

}

* 21 -

оператор отношения != означает “не равно”.

Основная проблема заключается в том, чтобы зафиксиро-

вать конец файла ввода. Обычно, когда функция GETCHAR натал-

кивается на конец файла ввода, она возвращает значение , не

являющееся действительным символом; таким образом, программа

может установить, что файл ввода исчерпан. Единственное ос-

ложнение, являющееся значительным неудобством, заключается в

существовании двух общеупотребительных соглашений о том, ка-

кое значение фактически является признаком конца файла. Мы

отсрочим решение этого вопроса, использовав символическое

имя EOF для этого значения, каким бы оно ни было. На практи-

ке EOF будет либо -1, либо 0, так что для правильной работы

перед программой должно стоять собственно либо

#DEFINE EOF -1

либо

#DEFINE EOF 0

Использовав символическую константу EOF для представле-

ния значения, возвращаемого функцией GETCHAR при выходе на

конец файла, мы обеспечили, что только одна величина в прог-

рамме зависит от конкретного численного значения.

Мы также описали переменную 'C' как INT , а не CHAR , с

тем чтобы она могла хранить значение, возвращаемое GETCHAR .

как мы увидим в главе 2, эта величина действительно INT, так

как она должна быть в состоянии в дополнение ко всем возмож-

ным символам представлять и EOF.

Программистом, имеющим опыт работы на “C”, программа

копирования была бы написана более сжато. В языке “C” любое

присваивание, такое как

C = GETCHAR()

может быть использовано в выражении; его значение - просто

значение, присваиваемое левой части. Если присваивание сим-

вола переменной 'C' поместить внутрь проверочной части опе-

ратора WHILE , то программа копирования файла запишется в

виде:

MAIN() /\* COPY INPUT TO OUTPUT; 2ND VERSION \*/

{

INT C;

WHILE ((C = GETCHAR()) != EOF)

PUTCHAR©;

}

Программа извлекает символ , присваивает его переменной

'C' и затем проверяет, не является ли этот символ признаком

конца файла. Если нет - выполняется тело оператора WHILE,

выводящее этот символ. Затем цикл WHILE повторяется. когда,

наконец, будет достигнут конец файла ввода, оператор WHILE

завершается, а вместе с ним заканчивается выполнение и функ-

ции MAIN .

* 22 -

В этой версии централизуется ввод - в программе только

одно обращение к функции GETCHAR - и ужимается программа.

Вложение присваивания в проверяемое условие - это одно из

тех мест языка “C”, которое приводит к значительному сокра-

щению программ. Однако, на этом пути можно увлечься и начать

писать недоступные для понимания программы. Эту тенденцию мы

будем пытаться сдерживать.

Важно понять , что круглые скобки вокруг присваивания в

условном выражении действительно необходимы. Старшинство

операции != выше, чем операции присваивания =, а это означа-

ет, что в отсутствие круглых скобок проверка условия != бу-

дет выполнена до присваивания =. Таким образом, оператор

C = GETCHAR() != EOF

эквивалентен оператору

C = (GETCHAR() != EOF)

Это, вопреки нашему желанию, приведет к тому, что 'C'

будет принимать значение 0 или 1 в зависимости от того, на-

толкнется или нет GETCHAR на признак конца файла. Подробнее

об этом будет сказано в главе 2/.

1.5.3. Подсчет символов.

Следующая программа подсчитывает число символов; она

представляет собой небольшое развитие программы копирования.

MAIN() /\* COUNT CHARACTERS IN INPUT \*/

{

LONG NC;

NC = 0;

WHILE (GETCHAR() != EOF )

++NC;

PRINTF(“%1D\N”, NC);

}

Оператор

++NC;

демонстрирует новую операцию, ++, которая означает увеличе-

ние на единицу. Вы могли бы написать NC = NC + 1 , но ++NC

более кратко и зачастую более эффективно. Имеется соответст-

вующая операция—уменьшение на единицу. Операции ++ и—

могут быть либо префиксными (++NC), либо постфиксными

(NC++); эти две формы, как будет показано в главе 2, имеют в

выражениях различные значения, но как ++NC, так и NC++ уве-

личивают NC. Пока мы будем придерживаться префиксных опера-

ций.

* 23 -

Программа подсчета символов накапливает их количество в

переменной типа LONG, а не INT . На PDP-11 максимальное зна-

чение равно 32767, и если описать счетчик как INT , то он

будет переполняться даже при сравнительно малом файле ввода;

на языке “C” для HONEYWELL и IBM типы LONG и INT являются

синонимами и имеют значительно больший размер. Спецификация

преобразования %1D указывает PRINTF , что соответствующий

аргумент является целым типа LONG .

Чтобы справиться с еще большими числами, вы можете ис-

пользовать тип DOUBLE / FLOAT двойной длины/. мы также ис-

пользуем оператор FOR вместо WHILE с тем, чтобы проиллюстри-

ровать другой способ записи цикла.

MAIN() /\* COUNT CHARACTERS IN INPUT \*/

{

DOUBLE NC;

FOR (NC = 0; GETCHAR() != EOF; ++NC)

;

PRINTF(“%.0F\N”, NC);

}

Функция PRINTF использует спецификацию %F как для FLOAT

, так и для DOUBLE ; спецификация %.0F подавляет печать не-

существующей дробной части.

Тело оператора цикла FOR здесь пусто, так как вся рабо-

та выполняется в проверочной и реинициализационной частях.

Но грамматические правила языка “C” требуют, чтобы оператор

FOR имел тело. Изолированная точка с запятой, соответствую-

шая пустому оператору, появляется здесь, чтобы удовлетворить

этому требованию. Мы выделили ее на отдельную строку, чтобы

сделать ее более заметной.

Прежде чем мы распростимся с программой подсчета симво-

лов, отметим, что если файл ввода не содержит никаких симво-

лов, то условие в WHILE или FOR не выполнится при самом пер-

вом обращении к GETCHAR , и, следовательно , программа вы-

даст нуль, т.е. Правильный ответ. это важное замечание. од-

ним из приятных свойств операторов WHILE и FOR является то,

что они проверяют условие в начале цикла, т.е. До выполнения

тела. Если делать ничего не надо, то ничего не будет сдела-

но, даже если это означает, что тело цикла никогда не будет

выполняться. программы должны действовать разумно, когда они

обращаются с файлами типа “никаких символов”. Операторы

WHILE и FOR помогают обеспечить правильное поведение прог-

рамм при граничных значениях проверяемых условий.

1.5.4. Подсчет строк.

Следующая программа подсчитывает количество строк в

файле ввода. Предполагается, что строки ввода заканчиваются

символом новой строки \N, скрупулезно добавленным к каждой

выписанной строке.

* 24 -

MAIN() /\* COUNT LINES IN INPUT \*/

{

INT C,NL;

NL = 0;

WHILE ((C = GETCHAR()) != EOF)

IF (C =='\N')

++NL;

PRINTF(“%D\N”, NL);

}

Тело WHILE теперь содержит оператор IF , который в свою

очередь управляет оператором увеличения ++NL. Оператор IF

проверяет заключенное в круглые скобки условие и, если оно

истинно, выполняет следующий за ним оператор /или группу

операторов, заключенных в фигурные скобки/. Мы опять исполь-

зовали сдвиг вправо, чтобы показать, что чем управляет.

Удвоенный знак равенства == является обозначением в

языке “C” для “равно” /аналогично .EQ. В фортране/. Этот

символ введен для того, чтобы отличать проверку на равенство

от одиночного =, используемого при присваивании. Поскольку в

типичных “C” - программах знак присваивания встречается при-

мерно в два раза чаще, чем проверка на равенство, то естест-

венно, чтобы знак оператора был вполовину короче.

Любой отдельный символ может быть записан внутри оди-

ночных кавычек, и при этом ему соответствует значение, рав-

ное численному значению этого символа в машинном наборе сим-

волов; это называется символьной константой. Так, например,

'A' - символьная константа; ее значение в наборе символов

ASCII /американский стандартный код для обмена информацией/

равно 65, внутреннему представлению символа а. Конечно, 'A'

предпочтительнее, чем 65: его смысл очевиден и он не зависит

от конкретного машинного набора символов.

Условные последовательности, используемые в символьных

строках, также занимают законное место среди символьных кон-

стант. Так в проверках и арифметических выражениях '\N'

представляет значение символа новой строки. Вы должны твердо

уяснить, что '\N' - отдельный символ, который в выражениях

эквивалентен одиночному целому; с другой стороны “\N” - это

символьная строка, которая содержит только один символ. Воп-

рос о сопоставлении строк и символов обсуждается в главе 2.

Упражнение 1-6.

Напишите программу для подсчета пробелов, табуляций и

новых строк.

Упражнение 1-7.

Напишите программу, которая копирует ввод на вывод, за-

меняя при этом каждую последовательность из одного или более

пробелов на один пробел.

1.5.5. Подсчет слов.

Четвертая программа из нашей серии полезных программ

подсчитывает количество строк, слов и символов, используя

при этом весьма широкое определение, что словом является лю-

бая последовательность символов, не содержащая пробелов, та-

буляций или новых строк. /Это - упрощенная версия утилиты

'WC' системы 'UNIX'/

* 25 -

#DEFINE YES 1

#DEFINE NO 0

MAIN() /\* COUNT LINES, WORDS, CHARS IN INPUT \*/

{

INT C, NL, NW, INWORD;

INWORD = NO;

NL = NW = NC = 0;

WHILE((C = GETCHAR()) != EOF) {

++NC;

IF (C == '\N')

++NL;

IF (C==' ' \!\! C=='\N' \!\! C=='\T')

INWORD = NO;

ELSE IF (INWORD == NO) {

INWORD = YES;

++NW;

}

}

PRINTF(“%D %D %D\N”, NL, NW, NC);

}

Каждый раз, когда программа встречает первый символ

слова, она увеличивает счетчик числа слов на единицу. Пере-

менная INWORD следит за тем, находится ли программа в насто-

ящий момент внутри слова или нет; сначала этой переменной

присваивается “ не в слове”, чему соответствует значение NO.

Мы предпочитаем символические константы YES и NO литерным

значениям 1 и 0, потому что они делают программу более удоб-

ной для чтения. Конечно, в такой крошечной программе, как

эта, это не приводит к заметной разнице, но в больших прог-

раммах увеличение ясности вполне стоит тех скромных дополни-

тельных усилий, которых требует следование этому принципу с

самого начала. Вы также обнаружите, что существенные измене-

ния гораздо легче вносить в те программы, где числа фигури-

руют только в качестве символьных констант.

Строка

NL = NW = NC = 0;

полагает все три переменные равными нулю. Это не

особый случай, а следствие того обстоятельства, что операто-

ру присваивания соответствует некоторое значение и присваи-

вания проводятся последовательно справа налево. Таким обра-

зом, дело обстоит так, как если бы мы написали

NC = (NL = (NW = 0));

операция \!\! Означает OR , так что строка

IF( C==' ' \!\! C=='\N' \!\! C=='\T')

говорит “если с - пробел, или с - символ новой строки, или с

* табуляция ...”./условная последовательность \T является

изображением символа табуляции/.

* 26 -

Имеется соответствующая операция && для AND. Выражения,

связанные операциями && или \!\! , Рассматриваются слева на

право, и при этом гарантируется, что оценивание выражений

будет прекращено, как только станет ясно, является ли все

выражение истинным или ложным. Так, если 'C' оказывается

пробелом, то нет никакой необходимости проверять, является

ли 'C' символом новой строки или табуляции, и такие проверки

действительно не делаются. В данном случае это не имеет

принципиального значения, но, как мы скоро увидим, в более

сложных ситуациях эта особенность языка весьма существенна.

Этот пример также демонстрирует оператор ELSE языка “C”,

который указывает то действие, которое должно выполняться,

если условие, содержащееся в операторе IF, окажется ложным.

Общая форма такова:

IF (выражение)

оператор-1

ELSE оператор-2

Выполняется один и только один из двух операторов, свя-

занных с конструкцией IF-ELSE. Если выражение истинно, вы-

полняется оператор-1; если нет - выполняется оператор-2.

Фактически каждый оператор может быть довольно сложным. В

программе подсчета слов оператор, следующий за ELSE , явля-

ется опертором IF , который управляет двумя операторами в

фигурных скобках.

Упражнение 1-9.

Как бы вы стали проверять программу подсчета слов ?

Kакие имеются ограничения ?

Упражнение 1-10.

Напишите программу, которая будет печатать слова из фай-

ла ввода, причем по одному на строку.

Упражнение 1-11.

Переделайте программу подсчета слов, используя лучшее

пределение “слова”; считайте, например словом последователь-

ность букв, цифр и апострофов, рачинающуюся с буквы.

1.6. Массивы.

Давайте напишем программу подсчета числа появлений каж-

дой цифры, символов пустых промежутков/пробел, табуляции,

новая строка/ и всех остальных символов. Конечно, такая за-

дача несколько искусственна, но она позволит нам проиллюст-

рировать в одной программе сразу несколько аспектов языка

“C”.

Мы разбили вводимые символы на двенадцать категорий, и

нам удобнее использовать массив для хранения числа появлений

каждой цифры, а не десять отдельных переменных. Вот один из

вариантов программы:

* 27 -

MAIN() /\* COUNT DIGITS, WHITE SPACE, OTHERS \*/

{

INT C, I, NWHITE, NOTHER;

INT NDIGIT[10];

NWHITE = NOTHER = 0;

FOR (I = 0; I < 10; ++I)

NDIGIT[I] = 0;

WHILE ((C = GETCHAR()) != EOF)

IF (C >= '0' && C <= '9')

++NDIGIT[C-'0'];

ELSE IF(C== ' ' \!\! C== '\N' \!\! C== '\T')

++NWHITE;

ELSE

++NOTHER;

PRINTF(“DIGITS =”);

FOR (I = 0; I < 10; ++I)

PRINTF(“ %D”, NDIGIT[I]);

PRINTF(“\NWHITE SPACE = %D, OTHER = %D\N”,

NWHITE, NOTHER);

}

Описание

INT NDIGIT[10];

объявляет, что NDIGIT является массивом из десяти целых. в

языке “C” индексы массива всегда начинаются с нуля /а не с

1, как в фортране или PL/1/, так что элементами массива яв-

ляются NDIGIT[0], NDIGIT[1],..., NDIGIT[9]. эта особенность

отражена в циклах FOR , которые инициализируют и печатают

массив.

Индекс может быть любым целым выражением, которое, ко-

нечно, может включать целые переменные, такие как I , и це-

лые константы.

Эта конкретная программа сильно опирается на свойства

символьного представления цифр. Так, например, в программе

проверка

IF( C >= '0' && C <= '9')...

определяет, является ли символ в 'C' цифрой, и если это так,

то численное значение этой цифры определяется по формуле / C

* '0'/. Такой способ работает только в том случае, если зна-

чения символьных констант '0', '1' и т.д. Положительны, рас-

положены в порядке возрастания и нет ничего, кроме цифр,

между константами '0' и '9'. К счастью, это верно для всех

общепринятых наборов символов.

* 28 -

По определению перед проведением арифметических опера-

ций, вовлекающих переменные типа CHAR и INT, все они преоб-

разуются к типу INT, TAK что в арифметических выражениях пе-

ременные типа CHAR по существу идентичны переменным типа

INT. Это вполне естественно и удобно; например, C -'0'- это

целое выражение со значением между 0 и 9 в соответствии с

тем, какой символ от '0' до '9' хранится в 'C', и, следова-

тельно, оно является подходящим индексом для массива NDIGIT.

Выяснение вопроса, является ли данный символ цифрой,

символом пустого промежутка или чем-либо еще, осуществляется

последовательностью операторов

IF (C >= '0' && C <= '9')

++NDIGIT[C-'0'];

ELSE IF(C == ' ' \!\! C == '\N' \!\! C == '\T')

++NWHITE;

ELSE

++NOTHER;

Конструкция

IF (условие)

оператор

ELSE IF (условие)

оператор

ELSE

оператор

часто встречаются в программах как средство выражения ситуа-

ций, в которых осуществляется выбор одного из нескольких

возможных решений.

Программа просто движется сверху вниз до тех пор, пока

не удовлетворится какое-нибудь условие; тогда выполняется

соответствующий 'оператор', и вся конструкция завершается.

/Конечно, 'оператор' может состоять из нескольких операто-

ров, заключенных в фигурные скобки/. Если ни одно из условий

не удовлетворяется, то выполняется 'оператор', стоящий после

заключительного ELSE, если оно присутствует. Если последнеE

ELSE и соответствующий 'оператор' опущены (как в программе

подсчета слов), то никаких действий не производится. Между

начальным IF и конечным ELSE может помещаться произвольное

количество групп

ELSE IF (условие)

оператор

С точки зрения стиля целесообразно записывать эту конст-рукцию так, как мы показали, с тем чтобы длинные выражения не залезали за правый край страницы.

Оператор SWITCH (переключатель), который рассматривается

в главе 3, представляет другую возможность для записи раз-

ветвления на несколько вариантов. этот оператор особенно

удобен, когда проверяемое выражение является либо просто не-

которым целым, либо символьным выражением, совпадающим с од-

ной из некоторого набора констант. Версия этой программы,

использующая оператор SWITCH, будет для сравнения приведена

в главе 3.

* 29 -

Упражнение 1-12.

Напишите программу, печатающую гистограмму длин слов из

файла ввода. Самое легкое - начертить гистограмму горизон-

тально; вертикальная ориентация требует больших усилий.

1.7. Функции.

В языке “C” функции эквивалентны подпрограммам или функ-

циям в фортране или процедурам в PL/1, паскале и т.д. Функ-

ции дают удобный способ заключения некоторой части вычисле-

ний в черный ящик, который в дальнейшем можно использовать,

не интересуясь его внутренним содержанием. Использование

функций является фактически единственным способом справиться

с потенциальной сложностью больших программ. Если функции

организованы должным образом, то можно игнорировать то, как

делается работа; достаточно знание того, что делается. Язык

“C” разработан таким образом, чтобы сделать использование

функций легким, удобным и эффективным. Вам будут часто вст-

речаться функции длиной всего в несколько строчек, вызывае-

мые только один раз, и они используются только потому, что

это проясняет некоторую часть программы.

До сих пор мы использовали только предоставленные нам

функции типа PRINTF, GETCHAR и PUTCHAR; теперь пора написать

несколько наших собственных. так как в “C” нет операции воз-

ведения в степень, подобной операции \*\* в фортране или PL/1,

давайте проиллюстрируем механику определения функции на при-

мере функции POWER(M,N), возводящей целое м в целую положи-

тельную степень N. Так значение POWER(2,5) равно 32. Конеч-

но, эта функция не выполняет всей работы операции \*\*, пос-

кольку она действует только с положительными степенями не-

больших чисел, но лучше не создавать дополнительных затруд-

нений, смешивая несколько различных вопросов.

Ниже приводится функция POWER и использующая ее основная

программа, так что вы можете видеть целиком всю структуру.

MAIN() /\* TEST POWER FUNCTION \*/

{

INT I;

FOR(I = 0; I < 10; ++I)

PRINTF(“%D %D %D\N”,I,POWER(2,I),POWER(-3,I));

}

POWER(X,N) /\* RAISE X N-TH POWER; N > 0 \*/

INT X,N;

{

INT I, P;

P = 1;

FOR (I =1; I <= N; ++I)

P = P \* X;

RETURN (P);

}

* 30 -

Все функции имеют одинаковый вид:

имя (список аргументов, если они имеются)

описание аргументов, если они имеются

{

описания

операторы

}

Эти функции могут быть записаны в любом порядке и нахо-

диться в одном или двух исходных файлах. Конечно, если ис-

ходная программа размещается в двух файлах, вам придется

дать больше указаний при компиляции и загрузке, чем если бы

она находилась в одном, но это дело операционной системы, а

не атрибут языка. В данный момент, для того чтобы все полу-

ченные сведения о прогоне “C”- программ, не изменились в

дальнейшем, мы будем предполагать, что обе функции находятся

в одном и том же файле.

Функция POWER вызывается дважды в строке

PRINTF(“%D %D %D\N”,I,POWER(2,I),POWER(-3,I));

при каждом обращении функция POWER, получив два аргумента,

вазвращает целое значение, которое печатается в заданном

формате. В выражениях POWER(2,I) является точно таким же це-

лым, как 2 и I. /Не все функции выдают целое значение; мы

займемся этим вопросом в главе 4/.

Аргументы функции POWER должны быть описаны соответству-

ющим образом, так как их типы известны. Это сделано в строке

INT X,N;

которая следует за именем функции.

Описания аргументов помещаются между списком аргументов

и открывающейся левой фигурной скобкой; каждое описание за-

канчивается точкой с запятой. Имена, использованные для ар-

гументов функции POWER, являются чисто локальными и недос-

тупны никаким другим функциям: другие процедуры могут ис-

пользовать те же самые имена без возникновения конфликта.

Это верно и для переменных I и P; I в функции POWER никак не

связано с I в функции MAIN.

Значение, вычисленное функцией POWER, передаются в MAIN

с помощью оператора RETURN, точно такого же, как в PL/1.

внутри круглых скобок можно написать любое выражение. Функ-

ция не обязана возвращать какое-либо значение; оператор

RETURN, не содержащий никакого выражения, приводит к такой

же передаче управления, как “сваливание на конец” функции

при достижении конечной правой фигурной скобки, но при этом

в вызывающую функцию не возвращается никакого полезного зна-

чения.

Упражнение 1-13.

Апишите программу преобразования прописных букв из айла

ввода в строчные, используя при этом функцию OWER©, кото-

рая возвращает значение 'C', если C'- не буква, и значение

соответствующей строчной уквы, если 'C'-буква.

* 31 -

1.8. Аргументы - вызов по значению.

Один аспект в “C” может оказаться непривычным для прог-

раммистов, которые использовали другие языки, в частности,

фортран и PL/1. в языке “C” все аргументы функций передаются

“по значению”. это означает, что вызванная функция получает

значения своих аргументов с помощью временных переменных

/фактически через стек/, а не их адреса. Это приводит к не-

которым особенностям, отличным от тех, с которыми мы сталки-

вались в языках типа фортрана и PL/1, использующих “вызов по

ссылке “, где вызванная процедура работает с адресом аргу-

мента, а не с его значением.

Главное отличие состоит в том, что в “C” вызванная функ-

ция не может изменить переменную из вызывающей функции; она

может менять только свою собственную временную копию.

Вызов по значению, однако, не помеха, а весьма ценное

качество. Оно обычно приводит к более компактным программам,

содержащим меньше не относящихся к делу переменных, потому

что с аргументами можно обращаться как с удобно инициализи-

рованными локальными перемнными вызванной процедуры. Вот,

например, вариант функции POWER использующей это обстоятель-

ство

POWER(X,N) /\* RAISE X N-TH POWER; N > 0;

VERSION 2 \*/

INT X,N;

{

INT P;

FOR (P = 1; N > 0; --N)

P = P \* X;

RETURN (P);

}

Аргумент N используется как временная переменная; из не-

го вычитается единица до тех пор, пока он не станет нулем.

Переменная I здесь больше не нужна. чтобы ни происходило с N

внутри POWER это никак не влияет на аргумент, с которым пер-

воначально обратились к функции POWER.

При необходимости все же можно добиться, чтобы функция

изменила переменную из вызывающей программы. Эта программа

должна обеспечить установление адреса переменной /техничес-

ки, через указатель на переменную/, а в вызываемой функции

надо описать соответствующий аргумент как указатель и ссы-

латься к фактической переменной косвенно через него. Мы рас-

смотрим это подробно в главе 5.

Когда в качестве аргумента выступает имя массива, то

фактическим значением, передаваемым функции, является адрес

начала массива. /Здесь нет никакого копирования элементов

массива/. С помощью индексации и адреса начала функция может

найти и изменить любой элемент массива. Это - тема следующе-

го раздела.

* 32 -

1.9. Массивы символов.

По-видимому самым общим типом массива в “C” является

массив символов. Чтобы проиллюстрировать использование мас-

сивов символов и обрабатывающих их функций, давайте напишем

программу, которая читает набор строк и печатает самую длин-

ную из них. Основная схема программы достаточно проста:

WHILE (имеется еще строка)

IF (эта строка длиннее самой длинной из

предыдущих)

запомнить эту строку и ее длину

напечатать самую длинную строку

По этой схеме ясно, что программа естественным образом

распадается на несколько частей. Одна часть читает новую

строку, другая проверяет ее, третья запоминает, а остальные

части программы управляют этим процессом.

Поскольку все так прекрасно делится, было бы хорошо и

написать программу соответсвующим образом. Давайте сначала

напишем отдельную функцию GETLINE, которая будет извлекать

следующую строку из файла ввода; это - обобщение функции

GETCHAR. мы попытаемся сделать эту функцию по возможности

более гибкой, чтобы она была полезной и в других ситуациях.

Как минимум GETLINE должна передавать сигнал о возможном по-

явлении конца файла; более общий полезный вариант мог бы пе-

редавать длину строки или нуль, если встретится конец файла.

нуль не может быть длиной строки, так как каждая строка со-

держит по крайней мере один символ; даже строка, содержащая

только символ новой строки, имеет длину 1.

Когда мы находим строку, которая длиннее самой длинной

из предыдущих, то ее надо где-то запомнить. Это наводит на

мысль о другой функции, COPY , которая будет копировать но-

вую строку в место хранения.

Наконец, нам нужна основная программа для управления

функциями GETLINE и COPY . Вот результат :

#DEFINE MAXLINE 1000 /\* MAXIMUM INPUT

LINE SIZE \*/

MAIN() /\* FIND LONGEST LINE \*/

{

INT LEN; /\* CURRENT LINE LENGTH \*/

INT MAX; /\* MAXIMUM LENGTH SEEN SO FAR \*/

CHAR LINE[MAXLINE]; /\* CURRENT INPUT LINE \*/

CHAR SAVE[MAXLINE]; /\* LONGEST LINE, SAVED \*/

MAX = 0;

WHILE ((LEN = GETLINE(LINE, MAXLINE)) > 0)

IF (LEN > MAX) {

MAX = LEN;

COPY(LINE, SAVE);

}

IF (MAX > 0) /\* THERE WAS A LINE \*/

PRINTF(“%S”, SAVE);

}

* 33 -

GETLINE(S,LIM) /\* GET LINE INTO S,RETURN LENGTH \*/

CHAR S[];

INT LIM;

{

INT C, I;

FOR(I=0;I<LIM-1 && (C=GETCHAR())!=EOF && C!='\N';++I)

S[I] = C;

IF (C == '\N') {

S[I] = C;

++I;

}

S[I] = '\0';

RETURN(I);

}

COPY(S1, S2) /\* COPY S1 TO S2;

ASSUME S2 BIG ENOUGH \*/

CHAR S1[], S2[];

{

INT I;

I = 0;

WHILE ((S2[I] = S1[I] != '\0')

++I;

}

Функция MAIN и GETLINE общаются как через пару аргумен-

тов, так и через возвращаемое значение. аргументы GETLINE

описаны в строках

CHAR S[];

INT LIM;

которые указывают, что первый аргумент является массивом, а

второй - целым.

Длина массива S не указана, так как она определена в

MAIN . функция GETLINE использует оператор RETURN для пере-

дачи значения назад в вызывающую программу точно так же, как

это делала функция POWER. Одни функции возвращают некоторое

нужное значение; другие, подобно COPY, используются из-за их

действия и не возвращают никакого значения.

Чтобы пометить конец строки символов, функция GETLINE

помещает в конец создаваемого ей массива символ \0 /нулевой

символ, значение которого равно нулю/. Это соглашение ис-

пользуется также компилятором с языка “C”: когда в “C” -

программе встречается строчная константа типа

“HELLO\N”

* 34 -

то компилятор создает массив символов, содержащий символы

этой строки, и заканчивает его символом \0, с тем чтобы фун-

кции, подобные PRINTF, могли зафиксировать конец массива:

! H ! E ! L ! L ! O ! \N ! \0 !

Спецификация формата %S указывает, что PRINTF ожидает стро-

ку, представленную в такой форме. Проанализировав функцию

COPY, вы обнаружите, что и она опирается на тот факт, что ее

входной аргумент оканчивается символом \0, и копирует этот

символ в выходной аргумент S2. /Все это подразумевает, что

символ \0 не является частью нормального текста/.

Между прочим, стоит отметить, что даже в такой маленькой

программе, как эта, возникает несколько неприятных организа-

ционных проблем. Например, что должна делать MAIN, если она

встретит строку, превышающую ее максимально возможный раз-

мер? Функция GETLINE поступает разумно: при заполнении мас-

сива она прекращает дальнейшее извлечение символов, даже ес-

ли не встречает символа новой строки. Проверив полученную

длину и последний символ, функция MAIN может установить, не

была ли эта строка слишком длинной, и поступить затем, как

она сочтет нужным. Ради краткости мы опустили эту проблему.

Пользователь функции GETLINE никак не может заранее уз-

нать, насколько длинной окажется вводимая строка. Поэтому в

GETLINE включен контроль переполнения. в то же время пользо-

ватель функции COPY уже знает /или может узнать/, каков раз-

мер строк, так что мы предпочли не включать в эту функцию

дополнительный контроль.

Упражнение 1-14.

Переделайте ведущую часть программы поиска самой длинной

строки таким образом, чтобы она правильно печатала длины

сколь угодно длинных вводимых строк и возможно больший

текст.

Упржнение 1-15.

Напишите программу печати всех строк длиннее 80 симво-

лов.

Упражнение 1-16.

Напишите программу, которая будет удалять из каждой

строки стоящие в конце пробелы и табуляции, а также строки,

целиком состоящие из пробелов.

Упражнение 1-17.

Напишите функцию REVERSE(S), которая распологает сим-

вольную строку S в обратном порядке. С ее помощью напишите

программу, которая обратит каждую строку из файла ввода.

* 35 -

1.10. Область действия: внешние переменные.

Переменные в MAIN(LINE, SAVE и т.д.) являются внутренни-

ми или локальными по отношению к функции MAIN, потому что

они описаны внутри MAIN и никакая другая функция не имеет к

ним прямого доступа. Это же верно и относительно переменных

в других функциях; например, переменная I в функции GETLINE

никак не связана с I в COPY. Каждая локальная переменная су-

ществует только тогда, когда произошло обращение к соответс-

твующей функции, и исчезает, как только закончится выполне-

ние этой функции. По этой причине такие переменные, следуя

терминологии других языков, обычно называют автоматическими.

Мы впредь будем использовать термин автоматические при ссыл-

ке на эти динамические локальные переменные. /в главе 4 об-

суждается класс статической памяти, когда локальные перемен-

ные все же оказываются в состоянии сохранить свои значения

между обращениями к функциям/.

Поскольку автоматические переменные появляются и исчеза-

ют вместе с обращением к функции, они не сохраняют своих

значений в промежутке от одного вызова до другого, в силу

чего им при каждом входе нужно явно присваивать значения.

Если этого не сделать, то они будут содержать мусор.

В качестве альтернативы к автоматическим переменным мож-

но определить переменные, которые будут внешними для всех

функций, т.е. Глобальными переменными, к которым может обра-

титься по имени любая функция, которая пожелает это сделать.

(этот механизм весьма сходен с “COMMON” в фортране и

“EXTERNAL” в PL/1). Так как внешние переменные доступны всю-

ду, их можно использовать вместо списка аргументов для пере-

дачи данных между функциями. Кроме того, поскольку внешние

переменные существуют постоянно, а не появляются и исчезают

вместе с вызываемыми функциями, они сохраняют свои значения

и после того, как функции, присвоившие им эти значения, за-

вершат свою работу.

Внешняя переменная должна быть определена вне всех функ-

ций; при этом ей выделяется фактическое место в памяти. Та-

кая переменная должна быть также описана в каждой функции,

которая собирается ее использовать; это можно сделать либо

явным описанием EXTERN, либо неявным по контексту. Чтобы

сделать обсуждение более конкретным, давайте перепишем прог-

рамму поиска самой длинной строки, сделав LINE, SAVE и MAX

внешними переменными. Это потребует изменения описаний и тел

всех трех функций, а также обращений к ним.

#DEFINE MAXLINE 1000 /\* MAX. INPUT LINE SIZE\*/

CHAR LINE[MAXLINE]; /\* INPUT LINE \*/

CHAR SAVE[MAXLINE];/\* LONGEST LINE SAVED HERE\*/

INT MAX;/\*LENGTH OF LONGEST LINE SEEN SO FAR\*/

* 36 -

MAIN() /\*FIND LONGEST LINE; SPECIALIZED VERSION\*/

{

INT LEN;

EXTERN INT MAX;

EXTERN CHAR SAVE[];

MAX = 0;

WHILE ( (LEN = GETLINE()) > 0 )

IF ( LEN > MAX ) {

MAX = LEN;

COPY();

}

IF ( MAX > 0 ) /\* THERE WAS A LINE \*/

PRINTF( “%S”, SAVE );

}

GETLINE() /\* SPECIALIZED VERSION \*/

{

INT C, I;

EXTERN CHAR LINE[];

FOR (I = 0; I < MAXLINE-1

&& (C=GETCHAR()) !=EOF && C!='\N'; ++I)

LINE[I] = C;

++I;

}

LINE[I] = '\0'

RETURN(I)

}

COPY() /\* SPECIALIZED VERSION \*/

{

INT I;

EXTERN CHAR LINE[], SAVE[];

I = 0;

WHILE ((SAVE[I] = LINE[I]) !='\0')

++I;

}

Внешние переменные для функций MAIN, GETLINE и COPY оп-

ределены в первых строчках приведенного выше примера, кото-

рыми указывается их тип и вызывается отведение для них памя-

ти. синтаксически внешние описания точно такие же, как опи-

сания, которые мы использовали ранее, но так как они распо-

ложены вне функций, соответствующие переменные являются

внешними. Чтобы функция могла использовать внешнюю переме-

ную, ей надо сообщить ее имя. Один способ сделать это -

включить в функцию описание EXTERN; это описание отличается

от предыдущих только добавлением ключевого слова EXTERN.

* 37 -

В определенных ситуациях описание EXTERN может быть опу-

щено: если внешнее определение переменной находится в том же

исходном файле, раньше ее использования в некоторой конкрет-

ной функции, то не обязательно включать описание EXTERN для

этой переменной в саму функцию. Описания EXTERN в функциях

MAIN, GETLINE и COPY являются, таким образом, излишними.

Фактически, обычная практика заключается в помещении опреде-

лений всех внешних переменных в начале исходного файла и

последующем опускании всех описаний EXTERN.

Если программа находится в нескольких исходных файлах, и

некоторая переменная определена, скажем в файле 1, а исполь-

зуется в файле 2, то чтобы связать эти два вхождения пере-

менной, необходимо в файле 2 использовать описание EXTERN.

Этот вопрос подробно обсуждается в главе 4.

Вы должно быть заметили, что мы в этом разделе при ссыл-

ке на внешние переменные очень аккуратно используем слова

описание и определение. “Определение” относится к тому мес-

ту, где переменная фактически заводится и ей выделяется па-

мять; “описание” относится к тем местам, где указывается

природа переменной, но никакой памяти не отводится.

Между прочим, существует тенденция объявлять все, что ни

попадется, внешними переменными, поскольку кажется, что это

упрощает связи, - списки аргументов становятся короче и пе-

ременные всегда присутствуют, когда бы вам они ни понадоби-

лись. Но внешние переменные присутствуют и тогда, когда вы в

них не нуждаетесь. Такой стиль программирования чреват опас-

ностью, так как он приводит к программам, связи данных внут-

ри которых не вполне очевидны. Переменные при этом могут из-

меняться неожиданным и даже неумышленным образом, а програм-

мы становится трудно модифицировать, когда возникает такая

необходимость. Вторая версия программы поиска самой длинной

строки уступает первой отчасти по этим причинам, а отчасти

потому, что она лишила универсальности две весьма полезные

функции, введя в них имена переменных, с которыми они будут

манипулировать.

Упражнение 1-18.

Проверка в операторе FOR функции GETLINE довольно неук-

люжа. Перепишите программу таким образом, чтобы сделать эту

проверку более ясной, но сохраните при этом то же самое по-

ведение в конце файла и при переполнении буфера. Является ли

это поведение самым разумным?

1.11. Резюме

На данном этапе мы обсудили то, что можно бы назвать

традиционным ядром языка “C”. Имея эту горсть строительных

блоков, можно писать полезные программы весьма значительного

размера, и было бы вероятно неплохой идеей, если бы вы за-

держались здесь на какое-то время и поступили таким образом:

следующие ниже упражнения предлагают вам ряд программ нес-

колько большей сложности, чем те, которые были приведены в

этой главе.

* 38 -

После того как вы овладеете этой частью “C”, приступайте

к чтению следующих нескольких глав. Усилия, которые вы при

этом затратите, полностью окупятся, потому что в этих главах

обсуждаются именно те стороны “C”, где мощь и выразитель-

ность языка начинает становиться очевидной.

Упражнение 1-19.

Напишите программу DETAB, которая заменяет табуляции во

вводе на нужное число пробелов так, чтобы промежуток дости-

гал следующей табуляционной остановки. Предположите фиксиро-

ванный набор табуляционных остановок, например, через каждые

N позиций.

Упражнение 1-20.

Напишите программу ENTAB, которая заменяет строки пробе-

лов минимальным числом табуляций и пробелов, достигая при

этом тех же самых промежутков. Используйте те же табуляцион-

ные остановки, как и в DETAB.

Упражнение 1-21.

Напишите программу для “сгибания” длинных вводимых строк

после последнего отличного от пробела символа, стоящего до

столбца N ввода, где N - параметр. убедитесь, что ваша прог-

рамма делает что-то разумное с очень длинными строками и в

случае, когда перед указанным столбцом нет ни табуляций, ни

пробелов.

Упражнение 1-22.

Напишите программу удаления из “C”-программы всех ком-

ментариев. Не забывайте аккуратно обращаться с “закавыченны-

ми” строками и символьными константами.

Упражнение 1-23.

Напишите программу проверки “C”-программы на элементар-

ные синтаксические ошибки, такие как несоответствие круглых,

квадратных и фигурных скобок. Не забудьте о кавычках, как

одиночных, так и двойных, и о комментариях. (Эта программа

весьма сложна, если вы будете писать ее для самого общего

случая).

* 39 -

2. Типы, операции и выражения.

Переменные и константы являются основными объектами, с

которыми оперирует программа. Описания перечисляют перемен-

ные, которые будут использоваться, указывают их тип и, воз-

можно, их начальные значения. Операции определяют, что с ни-

ми будет сделано. выражения объединяют переменные и констан-

ты для получения новых значений. Все это - темы настоящей

главы.

2.1. Имена переменных.

Хотя мы этого сразу прямо не сказали, существуют некото-

рые ограничения на имена переменных и символических конс-

тант. Имена составляются из букв и цифр; первый символ дол-

жен быть буквой. Подчеркивание “\_” тоже считается буквой;

это полезно для удобочитаемости длинных имен переменных.

Прописные и строчные буквы различаются; традиционная практи-

ка в “с” - использовать строчные буквы для имен переменных,

а прописные - для символических констант.

Играют роль только первые восемь символов внутреннего

имени, хотя использовать можно и больше. Для внешних имен,

таких как имена функций и внешних переменных, это число мо-

жет оказаться меньше восьми, так как внешние имена использу-

ются различными ассемблерами и загрузчиками. Детали приво-

дятся в приложении а. Кроме того, такие ключевые слова как

IF, ELSE, INT, FLOAT и т.д., зарезервированы: вы не можете

использовать их в качестве имен переменных. (Они пишутся

строчными буквами).

Конечно, разумно выбирать имена переменных таким обра-

зом, чтобы они означали нечто, относящееся к назначению пе-

ременных, и чтобы было менее вероятно спутать их при написа-

нии.

2.2. Типы и размеры данных.

Языке “C” имеется только несколько основных типов дан-

ных:

CHAR один байт, в котором может находиться один символ из

внутреннего набора символов.

INT Целое, обычно соответствующее естественному размеру це-

лых в используемой машине.

FLOAT С плавающей точкой одинарной точности.

DOUBLE С плавающей точкой двойной точности.

Кроме того имеется ряд квалификаторов, которые можно ис-

пользовать с типом INT: SHORT (короткое), LONG (длинное) и

UNSIGNED (без знака). Квалификаторы SHORT и LONG указывают

на различные размеры целых. Числа без знака подчиняются за-

конам арифметики по модулю 2 в степени N, где N - число би-

тов в INT; числа без знаков всегда положительны. Описания с

квалификаторами имеют вид:

SHORT INT X;

LONG INT Y;

UNSIGNED INT Z;

* 40 -

Cлово INT в таких ситуациях может быть опущено, что

обычно и делается.

Количество битов, отводимых под эти объекты зависит от

имеющейся машины; в таблице ниже приведены некоторые харак-

терные значения.

Таблица 1

!

DEC PDP-11 HONEYWELL IBM 370 INTERDATA !

6000 8/32 !

!

ASCII ASCII EBCDIC ASCII !

!

CHAR 8-BITS 9-BITS 8-BITS 8-BITS !

INT 16 36 32 32 !

SHORT 16 36 16 16 !

LONG 32 36 32 32 !

FLOAT 32 36 32 32 !

DOUBLE 64 72 64 64 !

!

Цель состоит в том, чтобы SHORT и LONG давали возмож-

ность в зависимости от практических нужд использовать раз-

личные длины целых; тип INT отражает наиболее “естественный”

размер конкретной машины. Как вы видите, каждый компилятор

свободно интерпретирует SHORT и LONG в соответствии со свои-

ми аппаратными средствами. Все, на что вы можете твердо по-

лагаться, это то, что SHORT не длиннее, чем LONG.

2.3. Константы.

Константы типа INT и FLOAT мы уже рассмотрели. Отметим

еще только, что как обычная

123.456е-7,

так и “научная” запись

0.12е3

для FLOAT является законной.

Каждая константа с плавающей точкой считается имеющей

тип DOUBLE, так что обозначение “E” служит как для FLOAT,

так и для DOUBLE.

Длинные константы записываются в виде 123L. Обычная це-

лая константа, которая слишком длинна для типа INT, рассмат-

ривается как LONG.

* 41 -

Существует система обозначений для восьмеричных и шест-

надцатеричных констант: лидирующий 0(нуль) в константе типа

INT указывает на восьмеричную константу, а стоящие впереди

0X соответствуют шестнадцатеричной константе. Например, де-

сятичное число 31 можно записать как 037 в восьмеричной фор-

ме и как 0X1F в шестнадцатеричной. Шестнадцатеричные и вось-

меричные константы могут также заканчиваться буквой L, что

делает их относящимися к типу LONG.

2.3.1. Символьная константа.

Символьная константа - это один символ, заключенный в

одинарные кавычки, как, например, 'х'. Значением символьной

константы является численное значение этого символа во внут-

реннем машинном наборе символов. Например, в наборе символов

ASCII символьный нуль, или '0', имеет значение 48, а в коде

EBCDIC - 240, и оба эти значения совершенно отличны от числа

0. Написание '0' вместо численного значения, такого как 48

или 240, делает программу не зависящей от конкретного чис-

ленного представления этого символа в данной машине. Сим-

вольные константы точно так же участвуют в численных опера-

циях, как и любые другие числа, хотя наиболее часто они ис-

пользуются в сравнении с другими символами. Правила преобра-

зования будут изложены позднее.

Некоторые неграфические символы могут быть представлены

как символьные константы с помощью условных последователь-

ностей, как, например, \N (новая строка), \T (табуляция), \0

(нулевой символ), \\ (обратная косая черта), \' (одинарная

кавычка) и т.д. Хотя они выглядят как два символа, на самом

деле являются одним. Кроме того, можно сгенерировать произ-

вольную последовательность двоичных знаков размером в байт,

если написать

'\DDD'

где DDD - от одной до трех восьмеричных цифр, как в

#DEFINE FORMFEED '\014' /\* FORM FEED \*/

Символьная константа '\0', изображающая символ со значе-нием 0, часто записывается вместо целой константы 0 , чтобы подчеркнуть символьную природу некоторого выражения.

2.3.2. Константное выражение

Константное выражение - это выражение, состоящее из од-

них констант. Такие выражения обрабатываются во время компи-

ляции, а не при прогоне программы, и соответственно могут

быть использованы в любом месте, где можно использовать кон-

станту, как, например в

* 42 -

#DEFINE MAXLINE 1000

CHAR LINE[MAXLINE+1];

или

SECONDS = 60 \* 60 \* HOURS;

2.3.3. Строчная константа

Строчная константа - это последовательность, состоящая

из нуля или более символов, заключенных в двойные кавычки,

как, например,

“I AM A STRING” /\* я - строка \*/

или

“” /\* NULL STRING **/ /** нуль-строка \*/

Кавычки не являются частью строки, а служат только для

ее ограничения. те же самые условные последовательности, ко-

торые использовались в символьных константах, применяются и

в строках; символ двойной кавычки изображается как \”.

С технической точки зрения строка представляет собой

массив, элементами которого являются отдельные символы. Что-

бы программам было удобно определять конец строки, компиля-

тор автоматически помещает в конец каждой строки нуль-символ

\0. Такое представление означает, что не накладывается конк-

ретного ограничения на то, какую длину может иметь строка, и

чтобы определить эту длину, программы должны просматривать

строку полностью. При этом для физического хранения строки

требуется на одну ячейку памяти больше, чем число заключен-

ных в кавычки символов. Следующая функция STRLEN(S) вычисля-

ет длину символьной строки S не считая конечный символ \0.

STRLEN(S) /\* RETURN LENGTH OF S \*/

CHAR S[];

{

INT I;

I = 0;

WHILE (S[I] != '\0')

++I;

RETURN(I);

}

Будьте внимательны и не путайте символьную константу со

строкой, содержащей один символ: 'X' - это не то же самое,

что “X”. Первое - это отдельный символ, использованный с

целью получения численного значения, соответствующего букве

х в машинном наборе символов. Второе - символьная строка,

состоящая из одного символа (буква х) и \0.

* 43 -

2.4. Описания

Все переменные должны быть описаны до их использования,

хотя некоторые описания делаются неявно, по контексту. Опи-

сание состоит из спецификатора типа и следующего за ним

списка переменных, имеющих этот тип, как, например,

INT LOWER, UPPER, STEP;

CHAR C, LINE[1000];

Переменные можно распределять по описаниям любым обра-

зом; приведенные выше списки можно с тем же успехом записать

в виде

INT LOWER;

INT UPPER;

INT STEP;

CHAR C;

CHAR LINE[1000];

Такая форма занимает больше места, но она удобна для до-бавления комментария к каждому описанию и для последующих модификаций.

Переменным могут быть присвоены начальные значения внут-

ри их описания, хотя здесь имеются некоторые ограничения.

Если за именем переменной следуют знак равенства и констан-

та, то эта константа служит в качестве инициализатора, как,

например, в

CHAR BACKSLASH = '\\';

INT I = 0;

FLOAT EPS = 1.0E-5;

Если рассматриваемая переменная является внешней или

статической, то инициализация проводится только один раз,

согласно концепции до начала выполнения программы. Инициали-

зируемым явно автоматическим переменным начальные значения

присваиваются при каждом обращении к функции, в которой они

описаны. Автоматические переменные, не инициализируемые яв-

но, имеют неопределенные значения, (т.е. мусор). Внешние и

статические переменные по умолчанию инициализируются нулем,

но, тем не менее, их явная инициализация является признаком

хорошего стиля.

Мы продолжим обсуждение вопросов инициализации, когда

будем описывать новые типы данных.

2.5. Арифметические операции.

Бинарными арифметическими операциями являются +, -, \*, /

и операция деления по модулю %. Имеется унарная операция -,

но не существует унарной операции +.

* 44 -

При делении целых дробная часть отбрасывается. Выражение

X % Y

дает остаток от деления X на Y и, следовательно, равно нулю,

когда х делится на Y точно. Например, год является високос-

ным, если он делится на 4, но не делится на 100, исключая

то, что делящиеся на 400 годы тоже являются високосными. По-

этому

IF(YEAR % 4 == 0 && YEAR % 100 != 0 \!\! YEAR % 400 == 0)

год високосный

ELSE

год невисокосный

Операцию % нельзя использовать с типами FLOAT или

DOUBLE.

Операции + и - имеют одинаковое старшинство, которое

младше одинакового уровня старшинства операций \*, / и %, ко-

торые в свою очередь младше унарного минуса. Арифметические

операции группируются слева направо. (Сведения о старшинстве

и ассоциативности всех операций собраны в таблице в конце

этой главы). Порядок выполнения ассоциативных и коммутатив-

ных операций типа + и - не фиксируется; компилятор может пе-

регруппировывать даже заключенные в круглые скобки выраже-

ния, связанные такими операциями. таким образом, а+(B+C) мо-

жет быть вычислено как (A+B)+C. Это редко приводит к како-

му-либо расхождению, но если необходимо обеспечить строго

определенный порядок, то нужно использовать явные промежу-

точные переменные.

Действия, предпринимаемые при переполнении и антипере-

полнении (т.е. При получении слишком маленького по абсолют-

ной величине числа), зависят от используемой машины.

2.6. Операции отношения и логические операции

Операциями отношения являются

=> > =< <

все они имеют одинаковое старшинство. Непосредственно за ни-

ми по уровню старшинства следуют операции равенства и нера-

венства:

== !=

которые тоже имеют одинаковое старшинство. операции отноше-

ния младше арифметических операций, так что выражения типа

I<LIM-1 понимаются как I<(LIM-1), как и предполагается.

Логические связки && и \!\! более интересны. Выражения,

связанные операциями && и \!\!, вычисляются слева направо,

причем их рассмотрение прекращается сразу же как только ста-

новится ясно, будет ли результат истиной или ложью. учет

этих свойств очень существенен для написания правильно рабо-

тающих программ. Рассмотрим, например, оператор цикла в счи-

тывающей строку функции GETLINE, которую мы написали в главе

1.

* 45 -

FOR(I=0;I<LIM-1 && (C=GETCHAR()) != '\N' && C != EOF; ++I)

S[I]=C;

Ясно, что перед считыванием нового символа необходимо

проверить, имеется ли еще место в массиве S, так что условие

I<LIM-1 должно проверяться первым. И если это условие не вы-

полняется, мы не должны считывать следующий символ.

Так же неудачным было бы сравнение 'C' с EOF до обраще-

ния к функции GETCHAR : прежде чем проверять символ, его

нужно считать.

Старшинство операции && выше, чем у \!\!, и обе они

младше операций отношения и равенства. Поэтому такие выраже-

ния, как

I<LIM-1 && (C = GETCHAR()) != '\N' && C != EOF

не нуждаются в дополнительных круглых скобках. Но так как

операция != старше операции присваивания, то для достижения

правильного результата в выражении

(C = GETCHAR()) != '\N'

кобки необходимы.

Унарная операция отрицания ! Преобразует ненулевой или

истинный операнд в 0, а нулевой или ложный операнд в 1.

Обычное использование операции ! Заключается в записи

IF( ! INWORD )

Вместо

IF( INWORD == 0 )

Tрудно сказать, какая форма лучше. Конструкции типа ! INWORD

Читаются довольно удобно (“если не в слове”). Но в более

сложных случаях они могут оказаться трудными для понимания.

Упражнение 2-1.

Напишите оператор цикла, эквивалентный приведенному выше

оператору FOR, не используя операции &&.

2.7. Преобразование типов

Если в выражениях встречаются операнды различных типов,

то они преобразуются к общему типу в соответствии с неболь-

шим набором правил. В общем, автоматически производятся

только преобразования, имеющие смысл, такие как, например,

преобразование целого в плавающее в выражениях типа F+I. Вы-

ражения же, лишенные смысла, такие как использование пере-

менной типа FLOAT в качестве индекса, запрещены.

Во-первых, типы CHAR и INT могут свободно смешиваться в

арифметических выражениях: каждая переменная типа CHAR авто-

матически преобразуется в INT. Это обеспечивает значительную

гибкость при проведении определенных преобразований симво-

лов. Примером может служить функция ATOI, которая ставит в

соответствие строке цифр ее численный эквивалент.

* 46 -

ATOI(S) /\* CONVERT S TO INTEGER \*/

CHAR S[];

{

INT I, N;

N = 0;

FOR ( I = 0; S[I]>='0' && S[I]<='9'; ++I)

N = 10 \* N + S[I] - '0';

RETURN(N);

}

KAK Уже обсуждалось в главе 1, выражение

S[I] - '0'

имеет численное значение находящегося в S[I] символа, потому

что значение символов '0', '1' и т.д. образуют возрастающую

последовательность расположенных подряд целых положительных

чисел.

Другой пример преобразования CHAR в INT дает функция

LOWER, преобразующая данную прописную букву в строчную. Если

выступающий в качестве аргумента символ не является пропис-

ной буквой, то LOWER возвращает его неизменным. Приводимая

ниже программа справедлива только для набора символов ASCII.

LOWER© /\* CONVERT C TO LOWER CASE; ASCII ONLY \*/

INT C;

{

IF ( C >= 'A' && C <= 'Z' )

RETURN( C + '@' - 'A');

ELSE /\*@ Записано вместо 'A' строчного\*/

RETURN©;

}

Эта функция правильно работает при коде ASCII, потому что

численные значения, соответствующие в этом коде прописным и

строчным буквам, отличаются на постоянную величину, а каждый

алфавит является сплошным - между а и Z нет ничего, кроме

букв. Это последнее замечание для набора символов EBCDIC

систем IBM 360/370 оказывается несправедливым, в силу чего

эта программа на таких системах работает неправильно - она

преобразует не только буквы.

При преобразовании символьных переменных в целые возни-

кает один тонкий момент. Дело в том, что сам язык не указы-

вает, должны ли переменным типа CHAR соответствовать числен-

ные значения со знаком или без знака. Может ли при преобра-

зовании CHAR в INT получиться отрицательное целое? К сожале-

нию, ответ на этот вопрос меняется от машины к машине, отра-

жая расхождения в их архитектуре. На некоторых машинах

(PDP-11, например) переменная типа CHAR, крайний левый бит

которой содержит 1, преобразуется в отрицательное целое

(“знаковое расширение”). На других машинах такое преобразо-

вание сопровождается добавлением нулей с левого края, в ре-

зультате чего всегда получается положительное число.

* 47 -

Определение языка “C” гарантирует, что любой символ из

стандартного набора символов машины никогда не даст отрица-

тельного числа, так что эти символы можно свободно использо-

вать в выражениях как положительные величины. Но произволь-

ные комбинации двоичных знаков, хранящиеся как символьные

переменные на некоторых машинах, могут дать отрицательные

значения, а на других положительные.

Наиболее типичным примером возникновения такой ситуации

является сучай, когда значение 1 используется в качестве

EOF. Рассмотрим программу

CHAR C;

C = GETCHAR();

IF ( C == EOF )

...

На машине, которая не осуществляет знакового расширения,

переменная 'с' всегда положительна, поскольку она описана

как CHAR, а так как EOF отрицательно, то условие никогда не

выполняется. Чтобы избежать такой ситуации, мы всегда пре-

дусмотрительно использовали INT вместо CHAR для любой пере-

менной, получающей значение от GETCHAR.

Основная же причина использования INT вместо CHAR не

связана с каким-либо вопросом о возможном знаковом расшире-

нии. просто функция GETCHAR должна передавать все возможные

символы (чтобы ее можно было использовать для произвольного

ввода) и, кроме того, отличающееся значение EOF. Следова-

тельно значение EOF не может быть представлено как CHAR, а

должно храниться как INT.

Другой полезной формой автоматического преобразования

типов является то, что выражения отношения, подобные I>J, и

логические выражения, связанные операциями && и \!\!, по оп-

ределению имеют значение 1, если они истинны, и 0, если они

ложны. Таким образом, присваивание

ISDIGIT = C >= '0' && C <= '9';

полагает ISDIGIT равным 1, если с - цифра, и равным 0 в про-

тивном случае. (В проверочной части операторов IF, WHILE,

FOR и т.д. “Истинно” просто означает “не нуль”).

Неявные арифметические преобразования работают в основ-

ном, как и ожидается. В общих чертах, если операция типа +

или \*, которая связывает два операнда (бинарная операция),

имеет операнды разных типов, то перед выполнением операции

“низший” тип преобразуется к “высшему” и получается резуль-

тат “высшего” типа. Более точно, к каждой арифметической

операции применяется следующая последовательность правил

преобразования.

* Типы CHAR и SHORT преобразуются в INT, а FLOAT в

DOUBLE.

* 48 -
* Затем, если один из операндов имеет тип DOUBLE, то

другой преобразуется в DOUBLE, и результат имеет тип DOUBLE.

* В противном случае, если один из операндов имеет тип LONG, то другой преобразуется в LONG, и результат имеет тип LONG.
* В противном случае, если один из операндов имеет тип UNSIGNED, то другой преобразуется в UNSIGNED и результат имеет тип UNSIGNED.
* В противном случае операнды должны быть типа INT, и

результат имеет тип INT.

Подчеркнем, что все переменные типа FLOAT в выражениях пре-

образуются в DOUBLE; в “C” вся плавающая арифметика выполня-

ется с двойной точностью.

Преобразования возникают и при присваиваниях; значение

правой части преобразуется к типу левой, который и является

типом результата. Символьные переменные преобразуются в це-

лые либо со знаковым расширением ,либо без него, как описано

выше. Обратное преобразование INT в CHAR ведет себя хорошо -

лишние биты высокого порядка просто отбрасываются. Таким об-

разом

INT I;

CHAR C;

I = C;

C = I;

значение 'с' не изменяется. Это верно независимо от того,

вовлекается ли знаковое расширение или нет.

Если х типа FLOAT, а I типа INT, то как

х = I;

так и

I = х;

приводят к преобразованиям; при этом FLOAT преобразуется в

INT отбрасыванием дробной части. Тип DOUBLE преобразуется во

FLOAT округлением. Длинные целые преобразуются в более ко-

роткие целые и в переменные типа CHAR посредством отбрасыва-

ния лишних битов высокого порядка.

Так как аргумент функции является выражением, то при пе-

редаче функциям аргументов также происходит преобразование

типов: в частности, CHAR и SHORT становятся INT, а FLOAT

становится DOUBLE. Именно поэтому мы описывали аргументы

функций как INT и DOUBLE даже тогда, когда обращались к ним

с переменными типа CHAR и FLOAT.

Наконец, в любом выражении может быть осуществлено

(“принуждено”) явное преобразование типа с помощью конструк-

ции, называемой перевод (CAST). В этой конструкции, имеющей

вид

(имя типа) выражение

* 49 -

Выражение преобразуется к указанному типу по правилам

преобразования, изложенным выше. Фактически точный смысл

операции перевода можно описать следующим образом: выражение

как бы присваивается некоторой переменной указанного типа,

которая затем используется вместо всей конструкции. Напри-

мер, библиотечная процедура SQRT ожидает аргумента типа

DOUBLE и выдаст бессмысленный ответ, если к ней по небреж-

ности обратятся с чем-нибудь иным. таким образом, если N -

целое, то выражение

SQRT((DOUBLE) N)

до передачи аргумента функции SQRT преобразует N к типу

DOUBLE. (Отметим, что операция перевод преобразует значение

N в надлежащий тип; фактическое содержание переменной N при

этом не изменяется). Операция перевода имрация перевода име-

ет тот же уровень старшинства, что и другие унарные опера-

ции, как указывается в таблице в конце этой главы.

Упражнение 2-2.

Составьте программу для функции HTOI(S), которая преоб-

разует строку шестнадцатеричных цифр в эквивалентное ей це-

лое значение. При этом допустимыми цифрами являются цифры от

1 до 9 и буквы от а до F.

2.8. Операции увеличения и уменьшения

В языке “C” предусмотрены две необычные операции для

увеличения и уменьшения значений переменных. Операция увели-

чения ++ добавляет 1 к своему операнду, а операция уменьше-

ния—вычитает 1. Мы часто использовали операцию ++ для

увеличения переменных, как, например, в

IF(C == '\N')

++I;

Необычный аспект заключается в том, что ++ и—можно

использовать либо как префиксные операции (перед переменной,

как в ++N), либо как постфиксные (после переменной: N++).

Эффект в обоих случаях состоит в увеличении N. Но выражение

++N увеличивает переменную N до использования ее значения, в

то время как N++ увеличивает переменную N после того, как ее

значение было использовано. Это означает, что в контексте,

где используется значение переменной, а не только эффект

увеличения, использование ++N и N++ приводит к разным ре-

зультатам. Если N = 5, то

х = N++;

устанавливает х равным 5, а

х = ++N;

* 50 -

полагает х равным 6. В обоих случаях N становится равным 6.

Операции увеличения и уменьшения можно применять только к

переменным; выражения типа х=(I+J)++ являются незаконными.

В случаях, где нужен только эффект увеличения, а само

значение не используется, как, например, в

IF ( C == '\N' )

NL++;

выбор префиксной или постфиксной операции является делом

вкуса. но встречаются ситуации, где нужно использовать имен-

но ту или другую операцию. Рассмотрим, например, функцию

SQUEEZE(S,C), которая удаляет символ 'с' из строки S, каждый

раз, как он встречается.

SQUEEZE(S,C) /\* DELETE ALL C FROM S \*/

CHAR S[];

INT C;

{

INT I, J;

FOR ( I = J = 0; S[I] != '\0'; I++)

IF ( S[I] != C )

S[J++] = S[I];

S[J] = '\0';

}

Каждый раз, как встечается символ, отличный от 'с', он копи-

руется в текущую позицию J, и только после этого J увеличи-

вается на 1, чтобы быть готовым для поступления следующего

символа. Это в точности эквивалентно записи

IF ( S[I] != C ) {

S[J] = S[I];

J++;

}

Другой пример подобной конструкции дает функция GETLINE,

которую мы запрограммировали в главе 1, где можно заменить

IF ( C == '\N' ) {

S[I] = C;

++I;

}

более компактной записью

IF ( C == '\N' )

S[I++] = C;

* 51 -

В качестве третьего примера рассмотрим функцию

STRCAT(S,T), которая приписывает строку т в конец строки S,

образуя конкатенацию строк S и т. При этом предполагается,

что в S достаточно места для хранения полученной комбинации.

STRCAT(S,T) /\* CONCATENATE T TO END OF S \*/

CHAR S[], T[]; /\* S MUST BE BIG ENOUGH \*/

{

INT I, J;

I = J = 0;

WHILE (S[I] != '\0') / \*FIND END OF S \*/

I++;

WHILE((S[I++] = T[J++]) != '\0') /\*COPY T\*/

;

}

Tак как из T в S копируется каждый символ, то для подготовки

к следующему прохождению цикла постфиксная операция ++ при-

меняется к обеим переменным I и J.

Упражнение 2-3.

Напишите другой вариант функции SQUEEZE(S1,S2), который

удаляет из строки S1 каждый символ, совпадающий с каким-либо

символом строки S2.

Упражнение 2-4.

Напишите программу для функции ANY(S1,S2), которая нахо-

дит место первого появления в строке S1 какого-либо символа

из строки S2 и, если строка S1 не содержит символов строки

S2, возвращает значение -1.

2.9. Побитовые логические операции

В языке предусмотрен ряд операций для работы с битами;

эти операции нельзя применять к переменным типа FLOAT или

DOUBLE.

& Побитовое AND

\! Побитовое включающее OR

^ побитовое исключающее OR

<< сдвиг влево

>> сдвиг вправо

\^ дополнение (унарная операция)

“\” иммитирует вертикальную черту.

Побитовая операция AND часто используется для маскирования

некоторого множества битов; например, оператор

C = N & 0177

* 52 -

передает в 'с' семь младших битов N , полагая остальные рав-

ными нулю. Операция 'э' побитового OR используется для вклю-

чения битов:

C = X э MASK

устанавливает на единицу те биты в х , которые равны единице

в MASK.

Следует быть внимательным и отличать побитовые операции

& и 'э' от логических связок && и \!\! , Которые подразуме-

вают вычисление значения истинности слева направо. Например,

если х=1, а Y=2, то значение х&Y равно нулю , в то время как

значение X&&Y равно единице./почему?/

Операции сдвига << и >> осуществляют соответственно

сдвиг влево и вправо своего левого операнда на число битовых

позиций, задаваемых правым операндом. Таким образом , х<<2

сдвигает х влево на две позиции, заполняя освобождающиеся

биты нулями, что эквивалентно умножению на 4. Сдвиг вправо

величины без знака заполняет освобождающиеся биты на некото-

рых машинах, таких как PDP-11, заполняются содержанием зна-

кового бита /”арифметический сдвиг”/, а на других - нулем

/”логический сдвиг”/.

Унарная операция \^ дает дополнение к целому; это озна-

чает , что каждый бит со значением 1 получает значение 0 и

наоборот. Эта операция обычно оказывается полезной в выраже-

ниях типа

X & \^077

где последние шесть битов х маскируются нулем. Подчеркнем,

что выражение X&\^077 не зависит от длины слова и поэтому

предпочтительнее, чем, например, X&0177700, где предполага-

ется, что х занимает 16 битов. Такая переносимая форма не

требует никаких дополнительных затрат, поскольку \^077 явля-

ется константным выражением и, следовательно, обрабатывается

во время компиляции.

Чтобы проиллюстрировать использование некоторых операций

с битами, рассмотрим функцию GETBITS(X,P,N), которая возвра-

щает /сдвинутыми к правому краю/ начинающиеся с позиции р

поле переменной х длиной N битов. мы предполагаем , что

крайний правый бит имеет номер 0, и что N и р - разумно за-

данные положительные числа. например, GETBITS(х,4,3) возвра-

щает сдвинутыми к правому краю биты, занимающие позиции 4,3

и 2.

GETBITS(X,P,N) /\* GET N BITS FROM POSITION P \*/

UNSIGNED X, P, N;

{

RETURN((X >> (P+1-N)) & \^(\^0 << N));

}

* 53 -

Операция X >> (P+1-N) сдвигает желаемое поле в правый конец

слова. Описание аргумента X как UNSIGNED гарантирует, что

при сдвиге вправо освобождающиеся биты будут заполняться ну-

лями, а не содержимым знакового бита, независимо от того, на

какой машине пропускается программа. Все биты константного

выражения \^0 равны 1; сдвиг его на N позиций влево с по-

мощью операции \^0<<N создает маску с нулями в N крайних

правых битах и единицами в остальных; дополнение \^ создает

маску с единицами в N крайних правых битах.

Упражнение 2-5.

Переделайте GETBITS таким образом, чтобы биты отсчитыва-

лись слева направо.

Упражнение 2-6.

Напишите программу для функции WORDLENGTH(), вычисляющей

длину слова используемой машины, т.е. Число битов в перемен-

ной типа INT. Функция должна быть переносимой, т.е. Одна и

та же исходная программа должна правильно работать на любой

машине.

Упражнение 2-7.

Напишите программу для функции RIGHTROT(N,B), сдвигающей

циклически целое N вправо на B битовых позиций.

Упражнение 2-8.

Напишите программу для функции INVERT(X,P,N), которая

инвертирует (т.е. Заменяет 1 на 0 и наоборот) N битов X, на-

чинающихся с позиции P, оставляя другие биты неизмененными.

2.10. Операции и выражения присваивания.

Такие выражения, как

I = I + 2

в которых левая часть повторяется в правой части могут быть

записаны в сжатой форме

I += 2

используя операцию присваивания вида +=.

Большинству бинарных операций (операций подобных +, ко-

торые имеют левый и правый операнд) соответствует операция

присваивания вида оп=, где оп - одна из операций

+ - \* / % << >> & \^ \!

Если е1 и е2 - выражения, то

* 54 -

е1 оп= е2

эквивалентно

е1 = (е1) оп (е2)

за исключением того, что выражение е1 вычисляется только

один раз. Обратите внимание на круглые скобки вокруг е2:

X \*= Y + 1

то

X = X \* (Y + 1)

не

X = X \* Y + 1

В качестве примера приведем функцию BITCOUNT, которая подсчитывает число равных 1 битов у целого аргумента.

BITCOUNT(N) /\* COUNT 1 BITS IN N \*/

UNSIGNED N;

(

INT B;

FOR (B = 0; N != 0; N >>= 1)

IF (N & 01)

B++;

RETURN(B);

)

Не говоря уже о краткости, такие операторы приваивания

имеют то преимущество, что они лучше соответствуют образу

человеческого мышления. Мы говорим: “прибавить 2 к I” или

“увеличить I на 2”, но не “взять I, прибавить 2 и поместить

результат опять в I”. Итак, I += 2. Кроме того, в громоздких

выражениях, подобных

YYVAL[YYPV[P3+P4] + YYPV[P1+P2]] += 2

Tакая операция присваивания облегчает понимание программы,

так как читатель не должен скрупулезно проверять, являются

ли два длинных выражения действительно одинаковыми, или за-

думываться, почему они не совпадают. Такая операция присваи-

вания может даже помочь компилятору получить более эффектив-

ную программу.

Мы уже использовали тот факт, что операция присваивания

имеет некоторое значение и может входить в выражения; самый

типичный пример

* 55 -

WHILE ((C = GETCHAR()) != EOF)

присваивания, использующие другие операции присваивания (+=,

-= и т.д.) также могут входить в выражения, хотя это случа-

ется реже.

Типом выражения присваивания является тип его левого

операнда.

Упражнение 2-9.

В двоичной системе счисления операция X&(X-1) обнуляет

самый правый равный 1 бит переменной X.(почему?) используйте

это замечание для написания более быстрой версии функции

BITCOUNT.

2.11. Условные выражения.

Операторы

IF (A > B)

Z = A;

ELSE

Z = B;

конечно вычисляют в Z максимум из а и в. Условное выражение,

записанное с помощью тернарной операции “?:”, предоставляет

другую возможность для записи этой и аналогичных конструк-

ций. В выражении

е1 ? Е2 : е3

сначала вычисляется выражение е1. Если оно отлично от нуля

(истинно), то вычисляется выражение е2, которое и становится

значением условного выражения. В противном случае вычисляет-

ся е3, и оно становится значением условного выражения. Каж-

дый раз вычисляется только одно из выражения е2 и е3. Таким

образом, чтобы положить Z равным максимуму из а и в, можно

написать

Z = (A > B) ? A : B; /\* Z = MAX(A,B) \*/

Следует подчеркнуть, что условное выражение действитель-

но является выражением и может использоваться точно так же,

как любое другое выражение. Если е2 и е3 имеют разные типы,

то тип результата определяется по правилам преобразования,

рассмотренным ранее в этой главе. например, если F имеет тип

FLOAT, а N - тип INT, то выражение

(N > 0) ? F : N

Имеет тип DOUBLE независимо от того, положительно ли N или

нет.

* 56 -

Так как уровень старшинства операции ?: очень низок,

прямо над присваиванием, то первое выражение в условном вы-

ражении можно не заключать в круглые скобки. Однако, мы все

же рекомендуем это делать, так как скобки делают условную

часть выражения более заметной.

Использование условных выражений часто приводит к корот-

ким программам. Например, следующий ниже оператор цикла пе-

чатает N элементов массива, по 10 в строке, разделяя каждый

столбец одним пробелом и заканчивая каждую строку (включая

последнюю) одним символом перевода строки.

OR (I = 0; I < N; I++)

PRINTF(“%6D%C”,A[I],(I%10==9 \!\! I==N-1) ? '\N' : ' ')

Символ перевода строки записывается после каждого десятого

элемента и после N-го элемента. За всеми остальными элемен-

тами следует один пробел. Хотя, возможно, это выглядит муд-

реным, было бы поучительным попытаться записать это, не ис-

пользуя условного выражения.

Упражнение 2-10.

Перепишите программу для функции LOWER, которая переводит

прописные буквы в строчные, используя вместо конструкции

IF-ELSE условное выражение.

2.12. Старшинство и порядок вычисления.

В приводимой ниже таблице сведены правила старшинства и ас-

социативности всех операций, включая и те, которые мы еще не

обсуждали. Операции, расположенные в одной строке, имеют

один и тот же уровень старшинства; строки расположены в по-

рядке убывания старшинства. Так, например, операции \*, / и %

имеют одинаковый уровень старшинства, который выше, чем уро-

вень операций + и -.

OPERATOR ASSOCIATIVITY

() [] -> . LEFT TO RIGHT

! \^ ++ -- - (TYPE) \* & SIZEOF RIGHT TO LEFT

\* / % LEFT TO RIGHT

+ - LEFT TO RIGHT

<< >> LEFT TO RIGHT

< <= > >= LEFT TO RIGHT

* 57 -

== != LEFT TO RIGHT

& LEFT TO RIGHT

^ LEFT TO RIGHT

\! LEFT TO RIGHT

&& LEFT TO RIGHT

\!\! LEFT TO RIGHT

?: RIGHT TO LEFT

= += -= ETC. RIGHT TO LEFT

, (CHAPTER 3) LEFT TO RIGHT

Операции -> и . Используются для доступа к элементам струк-

тур; они будут описаны в главе 6 вместе с SIZEOF (размер

объекта). В главе 5 обсуждаются операции \* (косвенная адре-

сация) и & (адрес).

Отметим, что уровень старшинства побитовых логических опера-

ций &, ^ и э ниже уровня операций == и !=. Это приводит к

тому, что осуществляющие побитовую проверку выражения, по-

добные

IF ((X & MASK) == 0) ...

Для получения правильных результатов должны заключаться в

круглые скобки.

Как уже отмечалось ранее, выражения, в которые входит

одна из ассоциативных и коммутативных операций (\*, +, &, ^,

э), могут перегруппировываться, даже если они заключены в

круглые скобки. В большинстве случаев это не приводит к ка-

ким бы то ни было расхождениям; в ситуациях, где такие рас-

хождения все же возможны, для обеспечения нужного порядка

вычислений можно использовать явные промежуточные перемен-

ные.

В языке “C”, как и в большинстве языков, не фиксируется

порядок вычисления операндов в операторе. Например в опера-

торе вида

X = F() + G();

сначала может быть вычислено F, а потом G, и наоборот; поэ-

тому, если либо F, либо G изменяют внешнюю переменную, от

которой зависит другой операнд, то значение X может зависеть

от порядка вычислений. Для обеспечения нужной последователь-

ности промежуточные результаты можно опять запоминать во

временных переменных.

Подобным же образом не фиксируется порядок вычисления

аргументов функции, так что оператор

PRINTF(“%D %D\N”,++N,POWER(2,N));

* 58 -

может давать (и действительно дает) на разных машинах разные

результаты в зависимости от того, увеличивается ли N до или

после обращения к функции POWER. Правильным решением, конеч-

но, является запись

++N;

PRINTF(“%D %D\N”,N,POWER(2,N));

Обращения к функциям, вложенные операции присваивания,

операции увеличения и уменьшения приводят к так называемым

“побочным эффектам” - некоторые переменные изменяются как

побочный результат вычисления выражений. В любом выражении,

в котором возникают побочные эффекты, могут существовать

очень тонкие зависимости от порядка, в котором определяются

входящие в него переменные. примером типичной неудачной си-

туации является оператор

A[I] = I++;

Возникает вопрос, старое или новое значение I служит в ка-

честве индекса. Компилятор может поступать разными способами

и в зависимости от своей интерпретации выдавать разные ре-

зультаты. Тот случай, когда происходят побочные эффекты

(присваивание фактическим переменным), - оставляется на ус-

мотрение компилятора, так как наилучший порядок сильно зави-

сит от архитектуры машины.

Из этих рассуждений вытекает такая мораль: написание

программ, зависящих от порядка вычислений, является плохим

методом программирования на любом языке. Конечно, необходимо

знать, чего следует избегать, но если вы не в курсе, как не-

которые вещи реализованы на разных машинах, это неведение

может предохранить вас от неприятностей. (Отладочная прог-

рамма LINT укажет большинство мест, зависящих от порядка вы-

числений.

* 59 -

3. Поток управления

Управляющие операторы языка определяют порядок вычисле-

ний. В приведенных ранее примерах мы уже встречались с наи-

более употребительными управляющими конструкциями языка “C”;

здесь мы опишем остальные операторы управления и уточним

действия операторов, обсуждавшихся ранее.

3.1. Операторы и блоки

Такие выражения, как X=0, или I++, или PRINTF(...),

становятся операторами, если за ними следует точка с запя-

той, как, например,

X = 0;

I++;

PRINTF(...);

В языке “C” точка с запятой является признаком конца опера-

тора, а не разделителем операторов, как в языках типа алго-

ла.

Фигурные скобки /( и /) используются для объединения

описаний и операторов в составной оператор или блок, так что

они оказываются синтаксически эквивалентны одному оператору.

Один явный пример такого типа дают фигурные скобки, в кото-

рые заключаются операторы, составляющие функцию, другой -

фигурные скобки вокруг группы операторов в конструкциях IF,

ELSE, WHILE и FOR.(на самом деле переменные могут быть опи-

саны внутри любого блока; мы поговорим об этом в главе 4).

Точка с запятой никогда не ставится после первой фигурной

скобки, которая завершает блок.

3.2. IF - ELSE

Оператор IF - ELSE используется при необходимости сде-

лать выбор. Формально синтаксис имеет вид

IF (выражение)

оператор-1

ELSE

оператор-2,

Где часть ELSE является необязательной. Сначала вычисля-

ется выражение; если оно “истинно” /т.е. значение выражения

отлично от нуля/, то выполняется оператор-1. Если оно ложно

/значение выражения равно нулю/, и если есть часть с ELSE,

то вместо оператора-1 выполняется оператор-2.

* 60 -

Так как IF просто проверяет численное значение выраже-

ния, то возможно некоторое сокращение записи. Самой очевид-

ной возможностью является запись

IF (выражение)

вместо

IF (выражение !=0)

иногда такая запись является ясной и естественной, но време-

нами она становится загадочной.

То, что часть ELSE в конструкции IF - ELSE является нео-

бязательной, приводит к двусмысленности в случае, когда ELSE

опускается во вложенной последовательности операторов IF.

Эта неоднозначность разрешается обычным образом - ELSE свя-

зывается с ближайшим предыдущим IF, не содержащим ELSE.

Например, в

IF ( N > 0 )

IF( A > B )

Z = A;

ELSE

Z = B;

конструкция ELSE относится к внутреннему IF, как мы и пока-

зали, сдвинув ELSE под соответствующий IF. Если это не то,

что вы хотите, то для получения нужного соответствия необхо-

димо использовать фигурные скобки:

IF (N > 0) {

IF (A > B)

Z = A;

}

ELSE

Z = B;

Tакая двусмысленность особенно пагубна в ситуациях типа

IF (N > 0)

FOR (I = 0; I < N; I++)

IF (S[I] > 0) {

PRINTF(“...”);

RETURN(I);

}

ELSE /\* WRONG \*/

PRINTF(“ERROR - N IS ZERO\N”);

* 61 -

Запись ELSE под IF ясно показывает, чего вы хотите, но ком-

пилятор не получит соответствующего указания и свяжет ELSE с

внутренним IF. Ошибки такого рода очень трудно обнаруживают-

ся.

Между прочим, обратите внимание, что в

IF (A > B)

Z = A;

ELSE

Z = B;

после Z=A стоит точка с запятой. Дело в том, что согласно

грамматическим правилам за IF должен следовать оператор, а

выражение типа Z=A, являющееся оператором, всегда заканчива-

ется точкой с запятой.

3.3. ELSE - IF

Конструкция

IF (выражение)

оператор

ELSE IF (выражение)

оператор

ELSE IF (выражение)

оператор

ELSE

оператор

встречается настолько часто, что заслуживает отдельного

краткого рассмотрения. Такая последовательность операторов

IF является наиболее распространенным способом программиро-

вания выбора из нескольких возможных вариантов. выражения

просматриваются последовательно; если какое-то выражение

оказывается истинным,то выполняется относящийся к нему опе-

ратор, и этим вся цепочка заканчивается. Каждый оператор мо-

жет быть либо отдельным оператором, либо группой операторов

в фигурных скобках.

Последняя часть с ELSE имеет дело со случаем, когда ни

одно из проверяемых условий не выполняется. Иногда при этом

не надо предпринимать никаких явных действий; в этом случае

хвост

ELSE

оператор

может быть опущен, или его можно использовать для контроля,

чтобы засечь “невозможное” условие.

* 62 -

Для иллюстрации выбора из трех возможных вариантов при-

ведем программу функции, которая методом половинного деления

определяет, находится ли данное значение х в отсортированном

массиве V. Элементы массива V должны быть расположены в по-

рядке возрастания. Функция возвращает номер позиции (число

между 0 и N-1), в которой значение х находится в V, и -1,

если х не содержится в V.

BINARY(X, V, N) /\* FIND X IN V[0]...V[N-1] \*/

INT X, V[], N;

{

INT LOW, HIGH, MID;

LOW = 0;

HIGH = N - 1;

WHILE (LOW <= HIGH) {

MID = (LOW + HIGH) / 2;

IF (X < V[MID])

HIGH = MID - 1;

ELSE IF (X > V[MID])

LOW = MID + 1;

ELSE /\* FOUND MATCH \*/

RETURN(MID);

}

RETURN(-1);

}

Основной частью каждого шага алгоритма является провер-

ка, будет ли х меньше, больше или равен среднему элементу

V[MID]; использование конструкции ELSE - IF здесь вполне ес-

тественно.

3.4. Переключатель

Оператор SWITCH дает специальный способ выбора одного из

многих вариантов, который заключается в проверке совпадения

значения данного выражения с одной из заданных констант и

соответствующем ветвлении. В главе 1 мы привели программу

подсчета числа вхождений каждой цифры, символов пустых про-

межутков и всех остальных символов, использующую последова-

тельность IF...ELSE IF...ELSE. Вот та же самая программа с

переключателем.

* 63 -

MAIN() /\* COUNT DIGITS,WHITE SPACE, OTHERS \*/

{

INT C, I, NWHITE, NOTHER, NDIGIT[10];

NWHITE = NOTHER = 0;

FOR (I = 0; I < 10; I++)

NDIGIT[I] = 0;

WHILE ((C = GETCHAR()) != EOF)

SWITCH © {

CASE '0':

CASE '1':

CASE '2':

CASE '3':

CASE '4':

CASE '5':

CASE '6':

CASE '7':

CASE '8':

CASE '9':

NDIGIT[C-'0']++;

BREAK;

CASE ' ':

CASE '\N':

CASE '\T':

NWHITE++;

BREAK;

DEFAULT :

NOTHER++;

BREAK;

}

PRINTF(“DIGITS =”);

FOR (I = 0; I < 10; I++)

PRINTF(“ %D”, NDIGIT[I]);

PRINTF(“\NWHITE SPACE = %D, OTHER = %D\N”,

NWHITE, NOTHER);

Переключатель вычисляет целое выражение в круглых скоб-

ках (в данной программе - значение символа с) и сравнивает

его значение со всеми случаями (CASE). Каждый случай должен

быть помечен либо целым, либо символьной константой, либо

константным выражением. Если значение константного выраже-

ния, стоящего после вариантного префикса CASE, совпадает со

значением целого выражения, то выполнение начинается с этого

случая. Если ни один из случаев не подходит, то выполняется

оператор после префикса DEFAULT. Префикс DEFAULT является

необязательным ,если его нет, и ни один из случаев не подхо-

дит, то вообще никакие действия не выполняются. Случаи и вы-

бор по умолчанию могут располагаться в любом порядке. Все

случаи должны быть различными.

* 64 -

Оператор BREAK приводит к немедленному выходу из перек-

лючателя. Поскольку случаи служат только в качестве меток,

то если вы не предпримите явных действий после выполнения

операторов, соответствующих одному случаю, вы провалитесь на

следующий случай. Операторы BREAK и RETURN являются самым

обычным способом выхода из переключателя. Как мы обсудим

позже в этой главе, оператор BREAк можно использовать и для

немедленного выхода из операторов цикла WHILE, FOR и DO.

Проваливание сквозь случаи имеет как свои достоинства,

так и недостатки. К положительным качествам можно отнести

то, что оно позволяет связать несколько случаев с одним дей-

ствием, как было с пробелом, табуляцией и новой строкой в

нашем примере. Но в то же время оно обычно приводит к необ-

ходимости заканчивать каждый случай оператором BREAK, чтобы

избежать перехода к следующему случаю. Проваливание с одного

случая на другой обычно бывает неустойчивым, так как оно

склонно к расщеплению при модификации программы. За исключе-

нием, когда одному вычислению соответствуют несколько меток,

проваливание следует использовать умеренно.

Заведите привычку ставить оператор BREAK после последне-

го случая (в данном примере после DEFAULT), даже если это не

является логически необходимым. В один прекрасный день, ког-

да вы добавите в конец еще один случай, эта маленькая мера

предосторожности избавит вас от неприятностей.

Упражнение 3-1.

Напишите программу для функции EXPAND(S, T), которая ко-

пирует строку S в т, заменяя при этом символы табуляции и

новой строки на видимые условные последовательности, как \N

и \т. используйте переключатель.

3.5. Циклы - WHILE и FOR

Мы уже сталкивались с операторами цикла WHILE и FOR. В

конструкции

WHILE (выражение)

оператор

вычисляется выражение. Если его значение отлично от нуля, то

выполняется оператор и выражение вычисляется снова. Этот

цикл продолжается до тех пор, пока значение выражения не

станет нулем, после чего выполнение программы продолжается с

места после оператора.

Оператор

FOR (выражение 1; выражение 2; выражение 3)

оператор

* 65 -

эквивалентен последовательности

выражение 1;

WHILE (выражение 2) {

оператор

выражение 3;

}

Грамматически все три компонента в FOR являются выражениями.

наиболее распространенным является случай, когда выражение 1

и выражение 3 являются присваиваниями или обращениями к фун-

кциям, а выражение 2 - условным выражением. любая из трех

частей может быть опущена, хотя точки с запятой при этом

должны оставаться. Если отсутствует выражение 1 или выраже-

ние 3, то оно просто выпадает из расширения. Если же отсутс-

твует проверка, выражение 2, то считается, как будто оно

всегда истинно, так что

FOR (;;) {

...

}

является бесконечным циклом, о котором предполагается, что

он будет прерван другими средствами (такими как BREAK или

RETURN).

Использовать ли WHILE или FOR - это, в основном дело

вкуса. Например в

WHILE ((C = GETCHAR())

== ' ' \!\! C == '\N' \!\! C == '\T')

; /\* SKIP WHITE SPACE CHARACTERS \*/

нет ни инициализации, ни реинициализации, так что цикл WHILе

выглядит самым естественным.

Цикл FOR, очевидно, предпочтительнее там, где имеется

простая инициализация и реинициализация, поскольку при этом

управляющие циклом операторы наглядным образом оказываются

вместе в начале цикла. Это наиболее очевидно в конструкции

FOR (I = 0; I < N; I++)

которая является идиомой языка “C” для обработки первых N

элементов массива, аналогичной оператору цикла DO в фортране

и PL/1. Аналогия, однако, не полная, так как границы цикла

могут быть изменены внутри цикла, а управляющая переменная

сохраняет свое значение после выхода из цикла, какова бы ни

была причина этого выхода. Поскольку компонентами FOR могут

быть произвольные выражения, они не ограничиваются только

арифметическими прогрессиями. Тем не менее является плохим

стилем включать в FOR вычисления, которые не относятся к уп-

равлению циклом, лучше поместить их в управляемые циклом

операторы.

* 66 -

В качестве большего по размеру примера приведем другой

вариант функции ATOI, преобразующей строку в ее численный

эквивалент. Этот вариант является более общим; он допускает

присутствие в начале символов пустых промежутков и знака +

или -. (В главе 4 приведена функция ATOF, которая выполняет

то же самое преобразование для чисел с плавающей точкой).

Общая схема программы отражает форму поступающих данных:

* пропустить пустой промежуток, если он имеется
* извлечь знак, если он имеется
* извлечь целую часть и преобразовать ее

Каждый шаг выполняет свою часть работы и оставляет все в

подготовленном состоянии для следующей части. Весь процесс

заканчивается на первом символе, который не может быть

частью числа.

ATOI(S) /\* CONVERT S TO INTEGER \*/

CHAR S[];

{

INT I, N, SIGN;

FOR(I=0;S[I]==' ' \!\!

S[I]=='\N' \!\! S[I]=='\T';I++)

; /\* SKIP WHITE SPACE \*/

SIGN = 1;

IF(S[I] == '+' \!\! S[I] == '-') /\* SIGN \*/

SIGN = (S[I++]=='+') ? 1 : - 1;

FOR( N = 0; S[I] >= '0' && S[I] <= '9'; I++)

N = 10 \* N + S[I] - '0';

RETURN(SIGN \* N);

}

Преимущества централизации управления циклом становятся

еще более очевидными, когда имеется несколько вложенных цик-

лов. Следующая функция сортирует массив целых чисел по мето-

ду шелла. основная идея сортировки по шеллу заключается в

том, что сначала сравниваются удаленные элементы, а не смеж-

ные, как в обычном методе сортировки. Это приводит к быстро-

му устранению большой части неупорядоченности и сокращает

последующую работу. Интервал между элементами постепенно

сокращается до единицы, когда сортировка фактически превра-

щается в метод перестановки соседних элементов.

* 67 -

SHELL(V, N) /\* SORT V[0]...V[N-1]

INTO INCREASING ORDER \*/

INT V[], N;

{

INT GAP, I, J, TEMP;

FOR (GAP = N/2; GAP > 0; GAP /= 2)

FOR (I = GAP; I < N; I++)

FOR (J=I-GAP; J>=0 && V[J]>V[J+GAP]; J-=GAP) {

TEMP = V[J];

V[J] = V[J+GAP];

V[J+GAP] = TEMP;

}

}

Здесь имеются три вложенных цикла. Самый внешний цикл управ-

ляет интервалом между сравниваемыми элементами, уменьшая его

от N/2 вдвое при каждом проходе, пока он не станет равным

нулю. Средний цикл сравнивает каждую пару элементов, разде-

ленных на величину интервала; самый внутренний цикл перес-

тавляет любую неупорядоченную пару. Так как интервал в конце

концов сводится к единице, все элементы в результате упоря-

дочиваются правильно. Отметим, что в силу общности конструк-

ции FOR внешний цикл укладывается в ту же самую форму, что и

остальные, хотя он и не является арифметической прогрессией.

Последней операцией языка “C” является запятая “,”, ко-

торая чаще всего используется в операторе FOR. Два выраже-

ния, разделенные запятой, вычисляются слева направо, причем

типом и значением результата являются тип и значение правого

операнда. Таким образом, в различные части оператора FOR

можно включить несколько выражений, например, для параллель-

ного изменения двух индексов. Это иллюстрируется функцией

REVERSE(S), которая располагает строку S в обратном порядке

на том же месте.

REVERSE(S) /\* REVERSE STRING S IN PLACE \*/

CHAR S[];

{

INT C, I, J;

FOR(I = 0, J = STRLEN(S) - 1; I < J; I++, J--) {

C = S[I];

S[I] = S[J];

S[J] = C;

}

}

Запятые, которые разделяют аргументы функций, переменные в

описаниях и т.д., не имеют отношения к операции запятая и не

обеспечивают вычислений слева направо.

* 68 -

Упражнение 3-2.

Составьте программу для функции EXPAND(S1,S2), которая

расширяет сокращенные обозначения вида а-Z из строки S1 в

эквивалентный полный список авс...XYZ в S2. Допускаются сок-

ращения для строчных и прописных букв и цифр. Будьте готовы

иметь дело со случаями типа а-в-с, а-Z0-9 и -а-Z. (Полезное

соглашение состоит в том, что символ -, стоящий в начале или

конце, воспринимается буквально).

3.6. Цикл DO - WHILE

Как уже отмечалось в главе 1, циклы WHILE и FOR обладают

тем приятным свойством, что в них проверка окончания осущес-

твляется в начале, а не в конце цикла. Третий оператор цикла

языка “C”, DO-WHILE, проверяет условие окончания в конце,

после каждого прохода через тело цикла; тело цикла всегда

выполняется по крайней мере один раз. Синтаксис этого опера-

тора имеет вид:

DO

оператор

WHILE (выражение)

Сначала выполняется оператор, затем вычисляется выражение.

Если оно истинно, то оператор выполняется снова и т.д. Если

выражение становится ложным, цикл заканчивается.

Как и можно было ожидать, цикл DO-WHILE используется

значительно реже, чем WHILE и FOR, составляя примерно пять

процентов от всех циклов. Тем не менее, иногда он оказывает-

ся полезным, как, например, в следующей функции ITOA, кото-

рая преобразует число в символьную строку (обратная функции

ATOI). Эта задача оказывается несколько более сложной, чем

может показаться сначала. Дело в том, что простые методы вы-

деления цифр генерируют их в неправильном порядке. Мы пред-

почли получить строку в обратном порядке, а затем обратить

ее.

* 69 -

ITOA(N,S) /\*CONVERT N TO CHARACTERS IN S \*/

CHAR S[];

INT N;

{

INT I, SIGN;

IF ((SIGN = N) < 0) /\* RECORD SIGN \*/

N = -N; /\* MAKE N POSITIVE \*/

I = 0;

DO { /\* GENERATE DIGITS IN REVERSE ORDER \*/

S[I++] = N % 10 + '0';/\* GET NEXT DIGIT \*/

} WHILE ((N /=10) > 0); /\* DELETE IT \*/

IF (SIGN < 0)

S[I++] = '-'

S[I] = '\0';

REVERSE(S);

}

Цикл DO-WHILE здесь необходим, или по крайней мере удобен,

поскольку, каково бы ни было значение N, массив S должен со-

держать хотя бы один символ. Мы заключили в фигурные скобки

один оператор, составляющий тело DO-WHILе, хотя это и не

обязательно, для того, чтобы торопливый читатель не принял

часть WHILE за начало оператора цикла WHILE.

Упражнение 3-3.

--------------

При представлении чисел в двоичном дополнительном коде

наш вариант ITOA не справляется с наибольшим отрицательным

числом, т.е. Со значением N рAвным -2 в степени м-1, где м -

размер слова. объясните почему. Измените программу так, что-

бы она правильно печатала это значение на любой машине.

Упражнение 3-4.

--------------

Напишите аналогичную функцию ITOB(N,S), которая преобра-

зует целое без знака N в его двоичное символьное представле-

ние в S. Запрограммируйте функцию ITOH, которая преобразует

целое в шестнадцатеричное представление.

Упражнение 3-5.

---------------

Напишите вариант Iтоа, который имеет три, а не два аргу-

мента. Третий аргумент - минимальная ширина поля; преобразо-

ванное число должно, если это необходимо, дополняться слева

пробелами, так чтобы оно имело достаточную ширину.

* 70 -

3.7. Оператор BREAK

Иногда бывает удобным иметь возможность управлять выхо-

дом из цикла иначе, чем проверкой условия в начале или в

конце. Оператор BRеак позволяет выйти из операторов FOR,

WHILE и DO до окончания цикла точно так же, как и из перек-

лючателя. Оператор BRеак приводит к немедленному выходу из

самого внутреннего охватывающего его цикла (или переключате-

ля).

Следующая программа удаляет хвостовые пробелы и табуля-

ции из конца каждой строки файла ввода. Она использует опе-

ратор BRеак для выхода из цикла, когда найден крайний правый

отличный от пробела и табуляции символ.

#DEFINE MAXLINE 1000

MAIN() /\* REMOVE TRAILING BLANKS AND TABS \*/

{

INT N;

CHAR LINE[MAXLINE];

WHILE ((N = GETLINE(LINE,MAXLINE)) > 0) {

WHILE (--N >= 0)

IF (LINE[N] != ' ' && LINE[N] != '\T'

&& LINE[N] != '\N')

BREAK;

LINE[N+1] = '\0';

PRINTF(“%S\N”,LINE);

}

}

Функция GETLINE возвращает длину строки. Внутренний цикл

начинается с последнего символа LINE (напомним, что—N

уменьшает N до использования его значения) и движется в об-

ратном направлении в поиске первого символа , который отли-

чен от пробела, табуляции или новой строки. Цикл прерывает-

ся, когда либо найден такой символ, либо N становится отри-

цательным (т.е., когда просмотрена вся строка). Советуем вам

убедиться, что такое поведение правильно и в том случае,

когда строка состоит только из символов пустых промежутков.

В качестве альтернативы к BRеак можно ввести проверку в

сам цикл:

WHILE ((N = GETLINE(LINE,MAXLINE)) > 0) {

WHILE (--N >= 0

&& (LINE[N] == ' ' \!\! LINE[N] == '\T'

\!\! LINE[N] == '\N'))

;

...

}

* 71 -

Это уступает предыдущему варианту, так как проверка стано-

вится труднее для понимания. Проверок, которые требуют пе-

реплетения &&, \!\!, ! И круглых скобок, по возможности сле-

дует избегать.

3.8. Оператор CONTINUE

Оператор CONTINUE родственен оператору BRеак, но исполь-

зуется реже; он приводит к началу следующей итерации охваты-

вающего цикла (FOR, WHILE, DO ). В циклах WHILE и DO это оз-

начает непосредственный переход к выполнению проверочной

части; в цикле FOR управление передается на шаг реинициали-

зации. (Оператор CONTINUE применяется только в циклах, но не

в переключателях. Оператор CONTINUE внутри переключателя

внутри цикла вызывает выполнение следующей итерации цикла).

В качестве примера приведем фрагмент, который обрабаты-

вает только положительные элементы массива а; отрицательные

значения пропускаются.

FOR (I = 0; I < N; I++) {

IF (A[I] < 0) /\* SKIP NEGATIVE ELEMENTS \*/

CONTINUE;

... /\* DO POSITIVE ELEMENTS \*/

}

Оператор CONTINUE часто используется, когда последующая

часть цикла оказывается слишком сложной, так что рассмотре-

ние условия, обратного проверяемому, приводит к слишком глу-

бокому уровню вложенности программы.

Упражнение 3-6.

Напишите программу копирования ввода на вывод, с тем ис-

ключением, что из каждой группы последовательных одинаковых

строк выводится только одна. (Это простой вариант утилиты

UNIQ систем UNIX).

3.9. Оператор GOTO и метки

В языке “C” предусмотрен и оператор GOTO, которым беско-

нечно злоупотребляют, и метки для ветвления. С формальной

точки зрения оператор GOTO никогда не является необходимым,

и на практике почти всегда можно обойтись без него. Мы не

использовали GOTO в этой книге.

Тем не менее, мы укажем несколько ситуаций, где оператор

GOTO может найти свое место. Наиболее характерным является

его использование тогда, когда нужно прервать выполнение в

некоторой глубоко вложенной структуре, например, выйти сразу

из двух циклов. Здесь нельзя непосредственно использовать

оператор BRеак, так как он прерывает только самый внутренний

цикл. Поэтому:

* 72 -

FOR ( ... )

FOR ( ... ) {

...

IF (DISASTER)

GOTO ERROR;

}

...

ERROR:

CLEAN UP THE MESS

Если программа обработки ошибок нетривиальна и ошибки могут

возникать в нескольких местах, то такая организация оказыва-

ется удобной. Метка имеет такую же форму, что и имя перемен-

ной, и за ней всегда следует двоеточие. Метка может быть

приписана к любому оператору той же функции, в которой нахо-

дится оператор GOTO.

В качестве другого примера рассмотрим задачу нахождения

первого отрицательного элемента в двумерном массиве. (Много-

мерные массивы рассматриваются в главе 5). Вот одна из воз-

можностей:

FOR (I = 0; I < N; I++)

FOR (J = 0; J < M; J++)

IF (V[I][J] < 0)

GOTO FOUND;

/\* DIDN'T FIND \*/

...

FOUND:

/\* FOUND ONE AT POSITION I, J \*/

...

Программа, использующая оператор GOTO, всегда может быть

написана без него, хотя, возможно, за счет повторения неко-

торых проверок и введения дополнительных переменных. Напри-

мер, программа поиска в массиве примет вид:

FOUND = 0;

FOR (I = 0; I < N && !FOUND; I++)

FOR (J = 0; J < M && !FOUND; J++)

FOUND = V[I][J] < 0;

IF (FOUND)

/\* IT WAS AT I-1, J-1 \*/

...

ELSE

/\* NOT FOUND \*/

...

Хотя мы не являемся в этом вопросе догматиками, нам все

же кажется, что если и нужно использовать оператор GOTO, то

весьма умеренно.

* 73 -

4. Функции и структура программ.

Функции разбивают большие вычислительные задачи на ма-

ленькие подзадачи и позволяют использовать в работе то, что

уже сделано другими, а не начинать каждый раз с пустого мес-

та. Соответствующие функции часто могут скрывать в себе де-

тали проводимых в разных частях программы операций, знать

которые нет необходимости, проясняя тем самым всю программу,

как целое, и облегчая мучения при внесении изменений.

Язык “C” разрабатывался со стремлением сделать функции

эффективными и удобными для использования; “C”-программы

обычно состоят из большого числа маленьких функций, а не из

нескольких больших. Программа может размещаться в одном или

нескольких исходных файлах любым удобным образом; исходные

файлы могут компилироваться отдельно и загружаться вместе

наряду со скомпилированными ранее функциями из библиотек. Мы

здесь не будем вдаваться в детали этого процесса, поскольку

они зависят от используемой системы.

Большинство программистов хорошо знакомы с “библиотечны-

ми” функциями для ввода и вывода /GETCHAR , PUTCHAR/ и для

численных расчетов /SIN, COS, SQRT/. В этой главе мы сообщим

больше о написании новых функций.

4.1. Основные сведения.

Для начала давайте разработаем и составим программу пе-

чати каждой строки ввода, которая содержит определенную ком-

бинацию символов. /Это - специальный случай утилиты GREP

системы “UNIX”/. Например, при поиске комбинации “THE” в на-

боре строк

NOW IS THE TIME

FOR ALL GOOD

MEN TO COME TO THE AID

OF THEIR PARTY

в качестве выхода получим

NOW IS THE TIME

MEN TO COME TO THE AID

OF THEIR PARTY

основная схема выполнения задания четко разделяется на три

части:

WHILE (имеется еще строка)

IF (строка содержит нужную комбинацию)

вывод этой строки

* 74 -

Конечно, возможно запрограммировать все действия в виде

одной основной процедуры, но лучше использовать естественную

структуру задачи и представить каждую часть в виде отдельной

функции. С тремя маленькими кусками легче иметь дело, чем с

одним большим, потому что отдельные не относящиеся к сущест-

ву дела детали можно включить в функции и уменьшить возмож-

ность нежелательных взаимодействий. Кроме того, эти куски

могут оказаться полезными сами по себе.

“Пока имеется еще строка” - это GETLINE, функция, кото-

рую мы запрограммировали в главе 1, а “вывод этой строки” -

это функция PRINTF, которую уже кто-то подготовил для нас.

Это значит, что нам осталось только написать процедуру для

определения, содержит ли строка данную комбинацию символов

или нет. Мы можем решить эту проблему, позаимствовав разра-

ботку из PL/1: функция INDEX(S,т) возвращает позицию, или

индекс, строки S, где начинается строка T, и -1, если S не

содержит т . В качестве начальной позиции мы используем 0, а

не 1, потому что в языке “C” массивы начинаются с позиции

нуль. Когда нам в дальнейшем понадобится проверять на совпа-

дение более сложные конструкции, нам придется заменить толь-

ко функцию INDEX; остальная часть программы останется той же

самой.

После того, как мы потратили столько усилий на разработ-

ку, написание программы в деталях не представляет затрудне-

ний. ниже приводится целиком вся программа, так что вы може-

те видеть, как соединяются вместе отдельные части. Комбина-

ция символов, по которой производится поиск, выступает пока

в качестве символьной строки в аргументе функции INDEX, что

не является самым общим механизмом. Мы скоро вернемся к об-

суждению вопроса об инициализации символьных массивов и в

главе 5 покажем, как сделать комбинацию символов параметром,

которому присваивается значение в ходе выполнения программы.

Программа также содержит новый вариант функции GETLINE; вам

может оказаться полезным сравнить его с вариантом из главы

1.

#DEFINE MAXLINE 1000

MAIN() /\* FIND ALL LINES MATCHING A PATTERN \*/

{

CHAR LINE[MAXLINE];

WHILE (GETLINE(LINE, MAXLINE) > 0)

IF (INDEX(LINE, “THE”) >= 0)

PRINTF(“%S”, LINE);

}

* 75 -

GETLINE(S, LIM) /\* GET LINE INTO S, RETURN LENGTH \*

CHAR S[];

INT LIM;

{

INT C, I;

I = 0;

WHILE(--LIM>0 && (C=GETCHAR()) != EOF && C != '\N')

S[I++] = C;

IF (C == '\N')

S[I++] = C;

S[I] = '\0';

RETURN(I);

}

INDEX(S,T) /\* RETURN INDEX OF T IN S,-1 IF NONE \*/

CHAR S[], T[];

{

INT I, J, K;

FOR (I = 0; S[I] != '\0'; I++) {

FOR(J=I, K=0; T[K] !='\0' && S[J] == T[K]; J++; K++)

;

IF (T[K] == '\0')

RETURN(I);

}

RETURN(-1);

}

Каждая функция имеет вид имя (список аргументов, если они

имеются) описания аргументов, если они имеются

{

описания и операторы , если они имеются

}

Как и указывается, некоторые части могут отсутство-

вать; минимальной функцией является

DUMMY () { }

которая не совершает никаких действий.

/Такая ничего не делающая функция иногда оказывается

удобной для сохранения места для дальнейшего развития прог-

раммы/. если функция возвращает что-либо отличное от целого

значения, то перед ее именем может стоять указатель типа;

этот вопрос обсуждается в следующем разделе.

* 76 -

Программой является просто набор определений отдельных

функций. Связь между функциями осуществляется через аргумен-

ты и возвращаемые функциями значения /в этом случае/; ее

можно также осуществлять через внешние переменные. Функции

могут располагаться в исходном файле в любом порядке, а сама

исходная программа может размещаться на нескольких файлах,

но так, чтобы ни одна функция не расщеплялась.

Оператор RETURN служит механизмом для возвращения зна-

чения из вызванной функции в функцию, которая к ней обрати-

лась. За RETURN может следовать любое выражение:

RETURN (выражение)

Вызывающая функция может игнорировать возвращаемое

значение, если она этого пожелает. Более того, после RETURN

может не быть вообще никакого выражения; в этом случае в вы-

зывающую программу не передается никакого значения. Управле-

ние также возвращется в вызывающую программу без передачи

какого-либо значения и в том случае, когда при выполнении мы

“проваливаемся” на конец функции, достигая закрывающейся

правой фигурной скобки. EСли функция возвращает значение из

одного места и не возвращает никакого значения из другого

места, это не является незаконным, но может быть признаком

каких-то неприятностей. В любом случае “значением” функции,

которая не возвращает значения, несомненно будет мусор. От-

ладочная программа LINT проверяет такие ошибки.

Механика компиляции и загрузки “C”-программ, располо-

женных в нескольких исходных файлах, меняется от системы к

системе. В системе “UNIX”, например, эту работу выполняет

команда 'CC', упомянутая в главе 1. Предположим, что три

функции находятся в трех различных файлах с именами MAIN.с,

GETLINE.C и INDEX.с . Тогда команда

CC MAIN.C GETLINE.C INDEX.C

компилирует эти три файла, помещает полученный настраиваемый объектный код в файлы MAIN.O, GETLINE.O и INDEX.O и загружа-ет их всех в выполняемый файл, называемый A.OUT .

Если имеется какая-то ошибка, скажем в MAIN.C, то этот

файл можно перекомпилировать отдельно и загрузить вместе с

предыдущими объектными файлами по команде

CC MAIN.C GETLIN.O INDEX.O

Команда 'CC' использует соглашение о наименовании с “.с” и “.о” для того, чтобы отличить исходные файлы от объектных.

Упражнение 4-1.

Составьте программу для функции RINDEX(S,T), которая

возвращает позицию самого правого вхождения т в S и -1, если

S не содержит T.

* 77 -

4.2. Функции, возвращающие нецелые значения.

До сих пор ни одна из наших программ не содержала како-

го-либо описания типа функции. Дело в том, что по умолчанию

функция неявно описывается своим появлением в выражении или

операторе, как, например, в

WHILE (GETLINE(LINE, MAXLINE) > 0)

Если некоторое имя, которое не было описано ранее, появ-

ляется в выражении и за ним следует левая круглая скобка, то

оно по контексту считается именем некоторой функции. Кроме

того, по умолчанию предполагается, что эта функция возвраща-

ет значение типа INT. Так как в выражениях CHAR преобразует-

ся в INT, то нет необходимости описывать функции, возвращаю-

щие CHAR. Эти предположения покрывают большинство случаев,

включая все приведенные до сих пор примеры.

Но что происходит, если функция должна возвратить значе-

ние какого-то другого типа ? Многие численные функции, такие

как SQRT, SIN и COS возвращают DOUBLE; другие специальные

функции возвращают значения других типов. Чтобы показать,

как поступать в этом случае, давайте напишем и используем

функцию ATоF(S), которая преобразует строку S в эквивалент-

ное ей плавающее число двойной точности. Функция ATоF явля-

ется расширением атоI, варианты которой мы написали в главах

2 и 3; она обрабатывает необязательно знак и десятичную точ-

ку, а также целую и дробную часть, каждая из которых может

как присутствовать, так и отсутствовать./эта процедура пре-

образования ввода не очень высокого качества; иначе она бы

заняла больше места, чем нам хотелось бы/.

Во-первых, сама ATоF должна описывать тип возвращаемого

ею значения, поскольку он отличен от INT. Так как в выраже-

ниях тип FLOAT преобразуется в DOUBLE, то нет никакого смыс-

ла в том, чтобы ATOF возвращала FLOAT; мы можем с равным ус-

пехом воспользоваться дополнительной точностью, так что мы

полагаем, что возвращаемое значение типа DOUBLE. Имя типа

должно стоять перед именем функции, как показывается ниже:

DOUBLE ATOF(S) /\* CONVERT STRING S TO DOUBLE \*/

CHAR S[];

{

DOUBLE VAL, POWER;

INT I, SIGN;

* 78 -

FOR(I=0; S[I]==' ' \!\! S[I]=='\N' \!\! S[I]=='\T'; I++)

; /\* SKIP WHITE SPACE \*/

SIGN = 1;

IF (S[I] == '+' \!\! S[I] == '-') /\* SIGN \*/

SIGN = (S[I++] == '+') ? 1 : -1;

FOR (VAL = 0; S[I] >= '0' && S[I] <= '9'; I++)

VAL = 10 \* VAL + S[I] - '0';

IF (S[I] == '.')

I++;

FOR (POWER = 1; S[I] >= '0' && S[I] <= '9'; I++) {

VAL = 10 \* VAL + S[I] - '0';

POWER \*= 10;

}

RETURN(SIGN \* VAL / POWER);

}

Вторым, но столь же важным, является то, что вызывающая

функция должна объявить о том, что ATOF возвращает значение,

отличное от INT типа. Такое объявление демонстрируется на

примере следующего примитивного настольного калькулятора

/едва пригодного для подведения баланса в чековой книжке/,

который считывает по одному числу на строку, причем это чис-

ло может иметь знак, и складывает все числа, печатая сумму

после каждого ввода.

#DEFINE MAXLINE 100

MAIN() /\* RUDIMENTARY DESK CALKULATOR \*/

{

DOUBLE SUM, ATOF();

CHAR LINE[MAXLINE];

SUM = 0;

WHILE (GETLINE(LINE, MAXLINE) > 0)

PRINTF(“\T%.2F\N”,SUM+=ATOF(LINE));

Оисание

DOUBLE SUM, ATOF();

говорит, что SUM является переменной типа DOUBLE , и что

ATOF является функцией, возвращающей значение типа DOUBLE .

Эта мнемоника означает, что значениями как SUM, так и

ATOF(...) являются плавающие числа двойной точности.

* 79 -

Если функция ATOF не будет описана явно в обоих местах,

то в “C” предполагается, что она возвращает целое значение,

и вы получите бессмысленный ответ. Если сама ATOF и обраще-

ние к ней в MAIN имеют несовместимые типы и находятся в од-

ном и том же файле, то это будет обнаружено компилятором. Но

если ATOF была скомпилирована отдельно /что более вероятно/,

то это несоответствие не будет зафиксировано, так что ATOF

будет возвращать значения типа DOUBLE, с которым MAIN будет

обращаться, как с INT , что приведет к бессмысленным резуль-

татам. /Программа LINT вылавливает эту ошибку/.

Имея ATOF, мы, в принципе, могли бы с ее помощью напи-

сать ATOI (преобразование строки в INT):

ATOI(S) /\* CONVERT STRING S TO INTEGER \*/

CHAR S[];

{

DOUBLE ATOF();

RETURN(ATOF(S));

}

Обратите внимание на структуру описаний и оператор RETURN.

Значение выражения в

RETURN (выражение)

всегда преобразуется к типу функции перед выполнением самого

возвращения. Поэтому при появлении в операторе RETURN значе-

ние функции атоF, имеющее тип DOUBLE, автоматически преобра-

зуется в INT, поскольку функция ATOI возвращает INT. (Как

обсуждалось в главе 2, преобразование значения с плавающей

точкой к типу INT осуществляется посредством отбрасывания

дробной части).

Упражнение 4-2.

Расширьте ATOF таким образом, чтобы она могла работать с

числами вида

123.45е-6

где за числом с плавающей точкой может следовать 'E' и пока-

затель экспоненты, возможно со знаком.

4.3. Еще об аргументах функций.

В главе 1 мы уже обсуждали тот факт , что аргументы фун-

кций передаются по значению, т.е. вызванная функция получает

свою временную копию каждого аргумента, а не его адрес. это

означает, что вызванная функция не может воздействовать на

исходный аргумент в вызывающей функции. Внутри функции каж-

дый аргумент по существу является локальной переменной, ко-

торая инициализируется тем значением, с которым к этой функ-

ции обратились.

* 80 -

Если в качестве аргумента функции выступает имя массива,

то передается адрес начала этого массива; сами элементы не

копируются. Функция может изменять элементы массива, исполь-

зуя индексацию и адрес начала. Таким образом, массив переда-

ется по ссылке. В главе 5 мы обсудим, как использование ука-

зателей позволяет функциям воздействовать на отличные от

массивов переменные в вызывающих функциях.

Между прочим, несуществует полностью удовлетворительного

способа написания переносимой функции с переменным числом

аргументов. Дело в том, что нет переносимого способа, с по-

мощью которого вызванная функция могла бы определить, сколь-

ко аргументов было фактически передано ей в данном обраще-

нии. Таким образом, вы, например, не можете написать дейст-

вительно переносимую функцию, которая будет вычислять макси-

мум от произвольного числа аргументов, как делают встроенные

функции MAX в фортране и PL/1.

Обычно со случаем переменного числа аргументов безопасно

иметь дело, если вызванная функция не использует аргументов,

которые ей на самом деле не были переданы, и если типы сог-

ласуются. Самая распространенная в языке “C” функция с пере-

менным числом - PRINTF . Она получает из первого аргумента

информацию, позволяющую определить количество остальных ар-

гументов и их типы. Функция PRINTF работает совершенно неп-

равильно, если вызывающая функция передает ей недостаточное

количество аргументов, или если их типы не согласуются с ти-

пами, указанными в первом аргументе. Эта функция не является

переносимой и должна модифицироваться при использовании в

различных условиях.

Если же типы аргументов известны, то конец списка аргу-

ментов можно отметить, используя какое-то соглашение; напри-

мер, считая, что некоторое специальное значение аргумента

(часто нуль) является признаком конца аргументов.

4.4. Внешние переменные.

Программа на языке “C” состоит из набора внешних объек-

тов, которые являются либо переменными, либо функциями. Тер-

мин “внешний” используется главным образом в противопостав-

ление термину “внутренний”, которым описываются аргументы и

автоматические переменные, определенные внурти функций.

Внешние переменные определены вне какой-либо функции и, та-

ким образом, потенциально доступны для многих функций. Сами

функции всегда являются внешними, потому что правила языка

“C” не разрешают определять одни функции внутри других. По

умолчанию внешние переменные являются также и “глобальными”,

так что все ссылки на такую переменную, использующие одно и

то же имя (даже из функций, скомпилированных независимо),

будут ссылками на одно и то же. В этом смысле внешние пере-

менные аналогичны переменным COмMON в фортране и EXTERNAL в

PL/1. Позднее мы покажем, как определить внешние переменные

и функции таким образом, чтобы они были доступны не глобаль-

но, а только в пределах одного исходного файла.

* 81 -

В силу своей глобальной доступности внешние переменные

предоставляют другую, отличную от аргументов и возвращаемых

значений, возможность для обмена данными между функциями.

Если имя внешней переменной каким-либо образом описано, то

любая функция имеет доступ к этой переменной, ссылаясь к ней

по этому имени.

В случаях, когда связь между функциями осуществляется с

помощью большого числа переменных, внешние переменные оказы-

ваются более удобными и эффективными, чем использование

длинных списков аргументов. Как, однако, отмечалось в главе

1, это соображение следует использовать с определенной осто-

рожностью, так как оно может плохо отразиться на структуре

программ и приводить к программам с большим числом связей по

данным между функциями.

Вторая причина использования внешних переменных связана

с инициализацией. В частности, внешние массивы могут быть

инициализированы а автоматические нет. Мы рассмотрим вопрос

об инициализации в конце этой главы.

Третья причина использования внешних переменных обуслов-

лена их областью действия и временем существования. Автома-

тические переменные являются внутренними по отношению к фун-

кциям; они возникают при входе в функцию и исчезают при вы-

ходе из нее. Внешние переменные, напротив, существуют посто-

янно. Они не появляютя и не исчезают, так что могут сохра-

нять свои значения в период от одного обращения к функции до

другого. В силу этого, если две функции используют некоторые

общие данные, причем ни одна из них не обращается к другой ,

то часто наиболее удобным оказывается хранить эти общие дан-

ные в виде внешних переменных, а не передавать их в функцию

и обратно с помощью аргументов.

Давайте продолжим обсуждение этого вопроса на большом

примере. Задача будет состоять в написании другой программы

для калькулятора, лучшей,чем предыдущая. Здесь допускаются

операции +,-,\*,/ и знак = (для выдачи ответа).вместо инфикс-

ного представления калькулятор будет использовать обратную

польскую нотацию,поскольку ее несколько легче реализовать.в

обратной польской нотации знак следует за операндами; инфик-

сное выражение типа

(1-2)\*(4+5)=

записывается в виде

12-45+\*=

круглые скобки при этом не нужны

* 82 -

Реализация оказывается весьма простой.каждый операнд по-

мещается в стек; когда поступает знак операции,нужное число

операндов (два для бинарных операций) вынимается,к ним при-

меняется операция и результат направляется обратно в

стек.так в приведенном выше примере 1 и 2 помещаются в стек

и затем заменяются их разностью, -1.после этого 4 и 5 вво-

дятся в стек и затем заменяются своей суммой,9.далее числа

* 1 и 9 заменяются в стеке на их произведение,равное -9.опе-рация = печатает верхний элемент стека, не удаляя его (так что промежуточные вычисления могут быть проверены).

Сами операции помещения чисел в стек и их извлечения

очень просты,но, в связи с включением в настоящую программу

обнаружения ошибок и восстановления,они оказываются доста-

точно длинными. Поэтому лучше оформить их в виде отдельных

функций,чем повторять соответствующий текст повсюду в прог-

рамме. Кроме того, нужна отдельная функция для выборки из

ввода следующей операции или операнда. Таким образом, струк-

тура программы имеет вид:

WHILE( поступает операция или операнд, а не конец

IF ( число )

поместить его в стек

еLSE IF ( операция )

вынуть операнды из стека

выполнить операцию

поместить результат в стек

ELSE

ошибка

Основной вопрос, который еще не был обсужден, заключает-

ся в том,где поместить стек, т. Е. Какие процедуры смогут

обращаться к нему непосредственно. Одна из таких возможнос-

тей состоит в помещении стека в MAIN и передачи самого стека

и текущей позиции в стеке функциям, работающим со стеком. Но

функции MAIN нет необходимости иметь дело с переменными, уп-

равляющими стеком; ей естественно рассуждать в терминах по-

мещения чисел в стек и извлечения их оттуда. В силу этого мы

решили сделать стек и связанную с ним информацию внешними

переменными , доступными функциям PUSH (помещение в стек) и

POP (извлечение из стека), но не MAIN.

Перевод этой схемы в программу достаточно прост. Ведущая

программа является по существу большим переключателем по ти-

пу операции или операнду; это, по-видимому, более характер-

ное применеие переключателя, чем то, которое было продемонс-

трировано в главе 3.

#DEFINE MAXOP 20 /\* MAX SIZE OF OPERAND, OPERАTOR \*

#DEFINE NUMBER '0' /\* SIGNAL THAT NUMBER FOUND \*/

#DEFINE TOOBIG '9' /\* SIGNAL THAT STRING IS TOO BIG \*

* 83 -

MAIN() /\* REVERSE POLISH DESK CALCULATOR \*/

/(

INT TUPE;

CHAR S[MAXOP];

DOUBLE OP2,ATOF(),POP(),PUSH();

WHILE ((TUPE=GETOP(S,MAXOP)) !=EOF);

SWITCH(TUPE) /(

CASE NUMBER:

PUSH(ATOF(S));

BREAK;

CASE '+':

PUSH(POP()+POP());

BREAK;

CASE '\*':

PUSH(POP()\*POP());

BREAK;

CASE '-':

OP2=POP();

PUSH(POP()-OP2);

BREAK;

CASE '/':

OP2=POP();

IF (OP2 != 0.0)

PUSH(POP()/OP2);

ELSE

PRINTF(“ZERO DIVISOR POPPED\N”);

BREAK;

CASE '=':

PRINTF(“\T%F\N”,PUSH(POP()));

BREAK;

CASE 'C':

CLEAR();

BREAK;

CASE TOOBIG:

PRINTF(“%.20S ... IS TOO LONG\N”,S)

BREAK;

/)

/)

#DEFINE MAXVAL 100 /\* MAXIMUM DEPTH OF VAL STACK \*/

* 84 -

INT SP = 0; /\* STACK POINTER \*/

DOUBLE VAL[MAXVAL]; /\*VALUE STACK \*/

DOUBLE PUSH(F) /\* PUSH F ONTO VALUE STACK \*/

DOUBLE F;

/(

IF (SP < MAXVAL)

RETURN(VAL[SP++] =F);

ELSE /(

PRINTF(“ERROR: STACK FULL\N”);

CLEAR();

RETURN(0);

/)

/)

DOUBLE POP() /\* POP TOP VALUE FROM STEACK \*/

/(

IF (SP > 0)

RETURN(VAL[--SP]);

ELSE /(

PRINTF(“ERROR: STACK EMPTY\N”);

CLEAR();

RETURN(0);

/)

/)

CLEAR() /\* CLEAR STACK \*/

/(

SP=0;

/)

Команда C очищает стек с помощью функции CLEAR, которая

также используется в случае ошибки функциями PUSH и POP. к

функции GETOP мы очень скоро вернемся.

Как уже говорилось в главе 1, переменная является внеш-

ней, если она определена вне тела какой бы то ни было функ-

ции. Поэтому стек и указатель стека, которые должны исполь-

зоваться функциями PUSH, POP и CLEAR, определены вне этих

трех функций. Но сама функция MAIN не ссылается ни к стеку,

ни к указателю стека - их участие тщательно замаскировано. В

силу этого часть программы, соответствующая операции = , ис-

пользует конструкцию

PUSH(POP());

для того, чтобы проанализировать верхний элемент стека, не

изменяя его.

Отметим также, что так как операции + и \* коммутативны,

порядок, в котором объединяются извлеченные операнды, несу-

щественен, но в случае операций - и / необходимо различать

левый и правый операнды.

* 85 -

Упражнение 4-3.

Приведенная основная схема допускает непосредственное

расширение возможностей калькулятора. Включите операцию де-

ления по модулю /%/ и унарный минус. Включите команду “сте-

реть”, которая удаляет верхний элемент стека. Введите коман-

ды для работы с переменными. /Это просто, если имена пере-

менных будут состоять из одной буквы из имеющихся двадцати

шести букв/.

4.5. Правила, определяющие область действия.

Функции и внешние переменные, входящие в состав

“C”-программы, не обязаны компилироваться одновременно;

программа на исходном языке может располагаться в нескольких

файлах, и ранее скомпилированные процедуры могут загружаться

из библиотек. Два вопроса представляют интерес:

Как следует составлять описания, чтобы переменные пра-

вильно воспринимались во время компиляции ?

Как следует составлять описания, чтобы обеспечить пра-

вильную связь частей программы при загрузке ?

4.5.1. Область действия.

Областью действия имени является та часть программы, в

которой это имя определено. Для автоматической переменной,

описанной в начале функции, областью действия является та

функция, в которой описано имя этой переменной, а переменные

из разных функций, имеющие одинаковое имя, считаются не от-

носящимися друг к другу. Это же справедливо и для аргументов

функций.

Область действия внешней переменной простирается от точ-

ки, в которой она объявлена в исходном файле, до конца этого

файла. Например, если VAL, SP, PUSH, POP и CLEAR определены

в одном файле в порядке, указанном выше, а именно:

INT SP = 0;

DOUBLE VAL[MAXVAL];

DOUBLE PUSH(F) {...}

DOUBLE POP() {...}

CLEAR() {...}

то переменные VAL и SP можно использовать в PUSH, POP и

CLEAR прямо по имени; никакие дополнительные описания не

нужны.

С другой стороны, если нужно сослаться на внешнюю пере-

менную до ее определения, или если такая переменная опреде-

лена в файле, отличном от того, в котором она используется,

то необходимо описание EXTERN.

* 86 -

Важно различать описание внешней переменной и ее опреде-

ление. описание указывает свойства переменной /ее тип, раз-

мер и т.д./; определение же вызывает еще и отведение памяти.

Если вне какой бы то ни было функции появляются строчки

INT SP;

DOUBLE VAL[MAXVAL];

то они определяют внешние переменные SP и VAL, вызывают от-

ведение памяти для них и служат в качестве описания для ос-

тальной части этого исходного файла. В то же время строчки

EXTERN INT SP;

EXTERN DOUBLE VAL[];

описывают в остальной части этого исходного файла переменную

SP как INT, а VAL как массив типа DOUBLE /размер которого

указан в другом месте/, но не создают переменных и не отво-

дят им места в памяти.

Во всех файлах, составляющих исходную программу, должно

содержаться только одно определение внешней переменной; дру-

гие файлы могут содержать описания EXTERN для доступа к ней.

/Описание EXTERN может иметься и в том файле, где находится

определение/. Любая инициализация внешней переменной прово-

дится только в определении. В определении должны указываться

размеры массивов, а в описании EXTERN этого можно не делать.

Хотя подобная организация приведенной выше программы и

маловероятна, но VAL и SP могли бы быть определены и инициа-

лизированы в одном файле, а функция PUSH, POP и CLEAR опре-

делены в другом. В этом случае для связи были бы необходимы

следующие определения и описания:

в файле 1:

INT SP = 0; /\* STACK POINTER \*/

DOUBLE VAL[MAXVAL]; /\* VALUE STACK \*/

в файле 2:

EXTERN INT SP;

EXTERN DOUBLE VAL[];

DOUBLE PUSH(F) {...}

DOUBLE POP() {...}

CLEAR() {...}

так как описания EXTERN 'в файле 1' находятся выше и вне

трех указанных функций, они относятся ко всем ним; одного

набора описаний достаточно для всего 'файла 2'.

* 87 -

Для программ большого размера обсуждаемая позже в этой

главе возможность включения файлов, #INCLUDE, позволяет

иметь во всей программе только одну копию описаний EXTERN и

вставлять ее в каждый исходный файл во время его компиляции.

Обратимся теперь к функции GETOP, выбирающей из файла

ввода следующую операцию или операнд. Основная задача прос-

та: пропустить пробелы, знаки табуляции и новые строки. Если

следующий символ отличен от цифры и десятичной точки, то

возвратить его. В противном случае собрать строку цифр /она

может включать десятичную точку/ и возвратить NUMBER как

сигнал о том, что выбрано число.

Процедура существенно усложняется, если стремиться пра-

вильно обрабатывать ситуацию, когда вводимое число оказыва-

ется слишком длинным. Функция GETOP считывает цифры подряд

/возможно с десятичной точкой/ и запоминает их, пока после-

довательность не прерывается. Если при этом не происходит

переполнения, то функция возвращает NUMBER и строку цифр.

Если же число оказывается слишком длинным, то GETOP отбрасы-

вает остальную часть строки из файла ввода, так что пользо-

ватель может просто перепечатать эту строку с места ошибки;

функция возвращает TOOBIG как сигнал о переполнении.

GETOP(S, LIM) /\* GET NEXT OPRERATOR OR OPERAND \*/

CHAR S[];

INT LIM;

{

INT I, C;

WHILE((C=GETCH())==' '\!\! C=='\T' \!\! C=='\N')

;

IF (C != '.' && (C < '0' \!\! C > '9'))

RETURN©;

S[0] = C;

FOR(I=1; (C=GETCHAR()) >='0' && C <= '9'; I++)

IF (I < LIM)

S[I] = C;

IF (C == '.') { /\* COLLECT FRACTION \*/

IF (I < LIM)

S[I] = C;

FOR(I++;(C=GETCHAR()) >='0' && C<='9';I++)

IF (I < LIM)

S[I] =C;

}

IF (I < LIM) { /\* NUMBER IS OK \*/

UNGETCH©;

S[I] = '\0';

RETURN (NUMBER);

} ELSE { /\* IT'S TOO BIG; SKIP REST OF LINE \*/

WHILE (C != '\N' && C != EOF)

C = GETCHAR();

S[LIM-1] = '\0';

RETURN (TOOBIG);

}

}

* 88 -

Что же представляют из себя функции 'GETCH' и 'UNGETCH'?

Часто так бывает, что программа, считывающая входные данные,

не может определить, что она прочла уже достаточно, пока она

не прочтет слишком много. Одним из примеров является выбор

символов, составляющих число: пока не появится символ, от-

личный от цифры, число не закончено. Но при этом программа

считывает один лишний символ, символ, для которого она еще

не подготовлена.

Эта проблема была бы решена, если бы было бы возможно

“прочесть обратно” нежелательный символ. Тогда каждый раз,

прочитав лишний символ, программа могла бы поместить его об-

ратно в файл ввода таким образом, что остальная часть прог-

раммы могла бы вести себя так, словно этот символ никогда не

считывался. к счастью, такое неполучение символа легко имми-

тировать, написав пару действующих совместно функций. Функ-

ция GETCH доставляет следующий символ ввода, подлежащий рас-

смотрению; функция UNGETCH помещает символ назад во ввод,

так что при следующем обращении к GETCH он будет возвращен.

То, как эти функции совместно работают, весьма просто.

Функция UNGETCH помещает возвращаемые назад символы в сов-

местно используемый буфер, являющийся символьным массивом.

Функция GETCH читает из этого буфера, если в нем что-либо

имеется; если же буфер пуст, она обращается к GETCHAR. При

этом также нужна индексирующая переменная, которая будет

фиксировать позицию текущего символа в буфере.

Так как буфер и его индекс совместно используются функ-

циями GETCH и UNGETCH и должны сохранять свои значения в пе-

риод между обращениями, они должны быть внешними для обеих

функций. Таким образом, мы можем написать GETCH, UNGETCH и

эти переменные как:

#DEFINE BUFSIZE 100

CHAR BUF[BUFSIZE]; /\* BUFFER FOR UNGETCH \*/

INT BUFP = 0; /\* NEXT FREE POSITION IN BUF \*/

GETCH() /\* GET A (POSSIBLY PUSHED BACK) CHARACTER \*/

{

RETURN((BUFP > 0) ? BUF[--BUFP] : GETCHAR());

}

UNGETCH© /\* PUSH CHARACTER BACK ON INPUT \*/

INT C;

{

IF (BUFP > BUFSIZE)

PRINTF(“UNGETCH: TOO MANY CHARACTERS\N”);

ELSE

BUF [BUFP++] = C;

}

Мы использовали для хранения возвращаемых символов массив, а

не отдельный символ, потому что такая общность может приго-

диться в дальнейшем.

* 89 -

Упражнение 4-4.

Напишите функцию UNGETS(S) , которая будет возвращать во

ввод целую строку. Должна ли UNGETS иметь дело с BUF и BUFP

или она может просто использовать UNGETCH ?

Упражнение 4-5.

Предположите, что может возвращаться только один символ. Из-

мените GETCH и UNGETCH соответствующим образом.

Упражнение 4-6.

Наши функции GETCH и UNGETCH не обеспечивают обработку возв-

ращенного символа EOF переносимым образом. Решите, каким

свойством должны обладать эти функции, если возвращается

EOF, и реализуйте ваши выводы.

4.6. Статические переменные.

Статические переменные представляют собой третий класс

памяти, в дополнении к автоматическим переменным и EXTERN, с

которыми мы уже встречались.

Статические переменные могут быть либо внутренними, либо

внешними. Внутренние статические переменные точно так же,

как и автоматические, являются локальными для некоторой фун-

кции, но, в отличие от автоматических, они остаются сущест-

вовать, а не появляются и исчезают вместе с обращением к

этой функции. это означает, что внутренние статические пере-

менные обеспечивают постоянное, недоступное извне хранение

внутри функции. Символьные строки, появляющиеся внутри функ-

ции, как, например, аргументы PRINTF , являются внутренними

статическими.

Внешние статические переменные определены в остальной

части того исходного файла, в котором они описаны, но не в

каком-либо другом файле. Таким образом, они дают способ

скрывать имена, подобные BUF и BUFP в комбинации

GETCH-UNGETCH, которые в силу их совместного использования

должны быть внешними, но все же не доступными для пользова-

телей GETCH и UNGETCH , чтобы исключалась возможность конф-

ликта. Если эти две функции и две переменные объеденить в

одном файле следующим образом

STATIC CHAR BUF[BUFSIZE]; /\* BUFFER FOR UNGETCH \*/

STATIC INT BUFP=0; /\*NEXT FREE POSITION IN BUF \*/

GETCH() {...}

UNGETCH() {...}

то никакая другая функция не будет в состоянии обратиться к

BUF и BUFP; фактически, они не будут вступать в конфликт с

такими же именами из других файлов той же самой программы.

Статическая память, как внутренняя, так и внешняя, спе-

цифицируется словом STATIC , стоящим перед обычным описани-

ем. Переменная является внешней, если она описана вне какой

бы то ни было функции, и внутренней, если она описана внутри

некоторой функции.

* 90 -

Нормально функции являются внешними объектами; их имена

известны глобально. возможно, однако, объявить функцию как

STATIC ; тогда ее имя становится неизвестным вне файла, в

котором оно описано.

В языке “C” “STATIC” отражает не только постоянство, но

и степень того, что можно назвать “приватностью”. Внутренние

статические объекты определены только внутри одной функции;

внешние статические объекты /переменные или функции/ опреде-

лены только внутри того исходного файла, где они появляются,

и их имена не вступают в конфликт с такими же именами пере-

менных и функций из других файлов.

Внешние статические переменные и функции предоставляют

способ организовывать данные и работающие с ними внутренние

процедуры таким образом, что другие процедуры и данные не

могут прийти с ними в конфликт даже по недоразумению. Напри-

мер, функции GETCH и UNGETCH образуют “модуль” для ввода и

возвращения символов; BUF и BUFP должны быть статическими,

чтобы они не были доступны извне. Точно так же функции PUSH,

POP и CLEAR формируют модуль обработки стека; VAR и SP тоже

должны быть внешними статическими.

4.7. Регистровые переменные.

Четвертый и последний класс памяти называется регистро-

вым. Описание REGISTER указывает компилятору, что данная пе-

ременная будет часто использоваться. Когда это возможно, пе-

ременные, описанные как REGISTER, располагаются в машинных

регистрах, что может привести к меньшим по размеру и более

быстрым программам. Описание REGISTER выглядит как

REGISTER INT X;

REGISTER CHAR C;

и т.д.; часть INT может быть опущена. Описание REGISTER мож-

но использовать только для автоматических переменных и фор-

мальных параметров функций. В этом последнем случае описания

выглядят следующим образом:

F(C,N)

REGISTER INT C,N;

{

REGISTER INT I;

...

}

* 91 -

На практике возникают некоторые ограничения на регистро-

вые переменные, отражающие реальные возможности имеющихся

аппаратных средств. В регистры можно поместить только нес-

колько переменных в каждой функции, причем только определен-

ных типов. В случае превышения возможного числа или исполь-

зования неразрешенных типов слово REGISTER игнорируется.

Кроме того невозможно извлечь адрес регистровой переменной

(этот вопрос обсуждается в главе 5). Эти специфические огра-

ничения варьируются от машины к машине. Так, например, на

PDP-11 эффективными являются только первые три описания

REGISTER в функции, а в качестве типов допускаются INT, CHAR

или указатель.

4.8. Блочная структура.

Язык “C” не является языком с блочной структурой в смыс-

ле PL/1 или алгола; в нем нельзя описывать одни функции

внутри других.

Переменные же, с другой стороны, могут определяться по

методу блочного структурирования. Описания переменных (вклю-

чая инициализацию) могут следовать за левой фигурной скоб-

кой,открывающей любой оператор, а не только за той, с кото-

рой начинается тело функции. Переменные, описанные таким об-

разом, вытесняют любые переменные из внешних блоков, имеющие

такие же имена, и остаются определенными до соответствующей

правой фигурной скобки. Например в

IF (N > 0) {

INT I; /\* DECLARE A NEW I \*/

FOR (I = 0; I < N; I++)

...

}

Областью действия переменной I является “истинная” ветвь

IF; это I никак не связано ни с какими другими I в програм-

ме.

Блочная структура влияет и на область действия внешних

переменных. Если даны описания

INT X;

F()

{

DOUBLE X;

...

}

То появление X внутри функции F относится к внутренней пере-

менной типа DOUBLE, а вне F - к внешней целой переменной.

это же справедливо в отношении имен формальных параметров:

INT X;

F(X)

DOUBLE X;

{

...

}

Внутри функции F имя X относится к формальному параметру, а

не к внешней переменной.

4.9. Инициализация.

Мы до сих пор уже много раз упоминали инициализацию, но

всегда мимоходом , среди других вопросов. Теперь, после того

как мы обсудили различные классы памяти, мы в этом разделе

просуммируем некоторые правила, относящиеся к инициализации.

Если явная инициализация отсутствует, то внешним и ста-

тическим переменным присваивается значение нуль; автомати-

ческие и регистровые переменные имеют в этом случае неопре-

деленные значения (мусор).

* 92 -

Простые переменные (не массивы или структуры) можно ини-

циализировать при их описании, добавляя вслед за именем знак

равенства и константное выражение:

INT X = 1;

CHAR SQUOTE = '\”;

LONG DAY = 60 \* 24; /\* MINUTES IN A DAY \*/

Для внешних и статических переменных инициализация выполня-

ется только один раз, на этапе компиляции. Автоматические и

регистровые переменные инициализируются каждый раз при входе

в функцию или блок.

В случае автоматических и регистровых переменных инициализа-

тор не обязан быть константой: на самом деле он может быть

любым значимым выражением, которое может включать определен-

ные ранее величины и даже обращения к функциям. Например,

инициализация в программе бинарного поиска из главы 3 могла

бы быть записана в виде

BINARY(X, V, N)

INT X, V[], N;

{

INT LOW = 0;

INT HIGH = N - 1;

INT MID;

...

}

вместо

BINARY(X, V, N)

INT X, V[], N;

{

INT LOW, HIGH, MID;

LOW = 0;

HIGH = N - 1;

...

}

По своему результату, инициализации автоматических перемен-

ных являются сокращенной записью операторов присваивания.

Какую форму предпочесть - в основном дело вкуса. мы обычно

используем явные присваивания, потому что инициализация в

описаниях менее заметна.

Автоматические массивы не могут быть инициализированы. Внеш-

ние и статические массивы можно инициализировать, помещая

вслед за описанием заключенный в фигурные скобки список на-

чальных значений, разделенных запятыми. Например программа

подсчета символов из главы 1, которая начиналась с

* 93 -

MAIN() /\* COUNT DIGITS, WHITE SPACE, OTHERS \*/

(

INT C, I, NWHITE, NOTHER;

INT NDIGIT[10];

NWHITE = NOTHER = 0;

FOR (I = 0; I < 10; I++)

NDIGIT[I] = 0;

...

)

Ожет быть переписана в виде

INT NWHITE = 0;

INT NOTHER = 0;

INT NDIGIT[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

MAIN() /\* COUNT DIGITS, WHITE SPACE, OTHERS \*/

(

INT C, I;

...

)

Эти инициализации фактически не нужны, так как все присваи-

ваемые значения равны нулю, но хороший стиль - сделать их

явными. Если количество начальных значений меньше, чем ука-

занный размер массива, то остальные элементы заполняются ну-

лями. Перечисление слишком большого числа начальных значений

является ошибкой. К сожалению, не предусмотрена возможность

указания, что некоторое начальное значение повторяется, и

нельзя инициализировать элемент в середине массива без пере-

числения всех предыдущих.

Для символьных массивов существует специальный способ

инициализации; вместо фигурных скобок и запятых можно ис-

пользовать строку:

CHAR PATTERN[] = “THE”;

Это сокращение более длинной, но эквивалентной записи:

CHAR PATTERN[] = { 'T', 'H', 'E', '\0' };

Если размер массива любого типа опущен, то компилятор опре-

деляет его длину, подсчитывая число начальных значений. В

этом конкретном случае размер равен четырем (три символа

плюс конечное \0).

* 94 -

4.10. Рекурсия.

В языке “C” функции могут использоваться рекурсивно; это

означает, что функция может прямо или косвенно обращаться к

себе самой. Традиционным примером является печать числа в

виде строки символов. как мы уже ранее отмечали, цифры гене-

рируются не в том порядке: цифры младших разрядов появляются

раньше цифр из старших разрядов, но печататься они должны в

обратном порядке.

Эту проблему можно решить двумя способами. Первый спо-

соб, которым мы воспользовались в главе 3 в функции ITOA,

заключается в запоминании цифр в некотором массиве по мере

их поступления и последующем их печатании в обратном поряд-

ке. Первый вариант функции PRINTD следует этой схеме.

PRINTD(N) /\* PRINT N IN DECIMAL \*/

INT N;

{

CHAR S[10];

INT I;

IF (N < 0) {

PUTCHAR('-');

N = -N;

}

I = 0;

DO {

S[I++] = N % 10 + '0'; /\* GET NEXT CHAR \*/

} WHILE ((N /= 10) > 0); /\* DISCARD IT \*/

WHILE (--I >= 0)

PUTCHAR(S[I]);

}

Альтернативой этому способу является рекурсивное реше-

ние, когда при каждом вызове функция PRINTD сначала снова

обращается к себе, чтобы скопировать лидирующие цифры, а за-

тем печатает последнюю цифру.

PRINTD(N) /\* PRINT N IN DECIMAL (RECURSIVE)\*/

INT N;

(

INT I;

IF (N < 0) {

PUTCHAR('-');

N = -N;

}

IF ((I = N/10) != 0)

PRINTD(I);

PUTCHAR(N % 10 + '0');

)

* 95 -

Когда функция вызывает себя рекурсивно, при каждом обра-

щении образуется новый набор всех автоматических переменных,

совершенно не зависящий от предыдущего набора. Таким обра-

зом, в PRINTD(123) первая функция PRINTD имеет N = 123. Она

передает 12 второй PRINTD, а когда та возвращает управление

ей, печатает 3. Точно так же вторая PRINTD передает 1

третьей (которая эту единицу печатает), а затем печатает 2.

Рекурсия обычно не дает никакой экономиии памяти, пос-

кольку приходится где-то создавать стек для обрабатываемых

значений. Не приводит она и к созданию более быстрых прог-

рамм. Но рекурсивные программы более компактны, и они зачас-

тую становятся более легкими для понимания и написания. Ре-

курсия особенно удобна при работе с рекурсивно определяемыми

структурами данных, например, с деревьями; хороший пример

будет приведен в главе 6.

Упражнение 4-7.

Приспособьте идеи, использованные в PRINTD для рекурсив-

ного написания ITOA; т.е. Преобразуйте целое в строку с по-

мощью рекурсивной процедуры.

Упражнение 4-8.

Напишите рекурсивный вариант функции REVERSE(S), которая

располагает в обратном порядке строку S.

4.11. Препроцессор языка “C”.

В языке “с” предусмотрены определенные расширения языка

с помощью простого макропредпроцессора. одним из самых расп-

ространенных таких расширений, которое мы уже использовали,

является конструкция #DEFINE; другим расширением является

возможность включать во время компиляции содержимое других

файлов.

4.11.1. Включение файлов

Для облегчения работы с наборами конструкций #DEFINE и

описаний (среди прочих средств) в языке “с” предусмотрена

возможность включения файлов. Любая строка вида

#INCLUDE “FILENAME”

заменяется содержимым файла с именем FILENAME. (Кавычки обя-

зательны). Часто одна или две строки такого вида появляются

в начале каждого исходного файла, для того чтобы включить

общие конструкции #DEFINE и описания EXTERN для глобальных

переменных. Допускается вложенность конструкций #INCLUDE.

Конструкция #INCLUDE является предпочтительным способом

связи описаний в больших программах. Этот способ гарантиру-

ет, что все исходные файлы будут снабжены одинаковыми опре-

делениями и описаниями переменных, и, следовательно, исклю-

чает особенно неприятный сорт ошибок. Естественно, когда ка-

кой-TO включаемый файл изменяется, все зависящие от него

файлы должны быть перекомпилированы.

* 96 -

4.11.2. Макроподстановка

Определение вида

#DEFINE TES 1

приводит к макроподстановке самого простого вида - замене

имени на строку символов. Имена в #DEFINE имеют ту же самую

форму, что и идентификаторы в “с”; заменяющий текст совер-

шенно произволен. Нормально заменяющим текстом является ос-

тальная часть строки; длинное определение можно продолжить,

поместив \ в конец продолжаемой строки. “Область действия”

имени, определенного в #DEFINE, простирается от точки опре-

деления до конца исходного файла. имена могут быть переопре-

делены, и определения могут использовать определения, сде-

ланные ранее. Внутри заключенных в кавычки строк подстановки

не производятся, так что если, например, YES - определенное

имя, то в PRINTF(“YES”) не будет сделано никакой подстанов-

ки.

Так как реализация #DEFINE является частью работы

маKропредпроцессора, а не собственно компилятора, имеется

очень мало грамматических ограничений на то, что может быть

определено. Так, например, любители алгола могут объявить

#DEFINE THEN

#DEFINE BEGIN {

#DEFINE END ;}

и затем написать

IF (I > 0) THEN

BEGIN

A = 1;

B = 2

END

Имеется также возможность определения макроса с аргумен-

тами, так что заменяющий текст будет зависеть от вида обра-

щения к макросу. Определим, например, макрос с именем MAX

следующим образом:

#DEFINE MAX(A, B) ((A) > (B) ? (A) : (B))

когда строка

X = MAX(P+Q, R+S);

будет заменена строкой

X = ((P+Q) > (R+S) ? (P+Q) : (R+S));

Такая возможность обеспечивает “функцию максимума”, которая

расширяется в последовательный код, а не в обращение к функ-

ции. При правильном обращении с аргументами такой макрос бу-

дет работать с любыми типами данных; здесь нет необходимости

в различных видах MAX для данных разных типов, как это было

бы с функциями.

* 97 -

Конечно, если вы тщательно рассмотрите приведенное выше

расширение MAX, вы заметите определенные недостатки. Выраже-

ния вычисляются дважды; это плохо, если они влекут за собой

побочные эффекты, вызванные, например, обращениями к функци-

ям или использованием операций увеличения. Нужно позаботить-

ся о правильном использовании круглых скобок, чтобы гаранти-

ровать сохранение требуемого порядка вычислений. (Рассмотри-

те макрос

#DEFINE SQUARE(X) X \* X

при обращении к ней, как SQUARE(Z+1)). Здесь возникают даже

некоторые чисто лексические проблемы: между именем макро и

левой круглой скобкой, открывающей список ее аргументов, не

должно быть никаких пробелов.

Тем не менее аппарат макросов является весьма ценным.

Один практический пример дает описываемая в главе 7 стандар-

тная библиотека ввода-вывода, в которой GETCHAR и PUTCHAR

определены как макросы (очевидно PUTCHAR должна иметь аргу-

мент), что позволяет избежать затрат на обращение к функции

при обработке каждого символа.

Другие возможности макропроцессора описаны в приложении

А.

Упражнение 4-9.

Определите макрос SWAP(X, Y), который обменивает значе-

ниями два своих аргумента типа INT. (В этом случае поможет

блочная структура).

* 98 -

5.Указатели и массивы

Указатель - это переменная, содержащая адрес другой пе-

ременной. указатели очень широко используются в языке “C”.

Это происходит отчасти потому, что иногда они дают единст-

венную возможность выразить нужное действие, а отчасти пото-

му, что они обычно ведут к более компактным и эффективным

программам, чем те, которые могут быть получены другими спо-

собами.

Указатели обычно смешивают в одну кучу с операторами

GOTO, характеризуя их как чудесный способ написания прог-

рамм, которые невозможно понять. Это безусловно спрAведливо,

если указатели используются беззаботно; очень просто ввести

указатели, которые указывают на что-то совершенно неожидан-

ное. Однако, при определенной дисциплине, использование ука-

зателей помогает достичь ясности и простоты. Именно этот ас-

пект мы попытаемся здесь проиллюстрировать.

5.1. Указатели и адреса

Так как указатель содержит адрес объекта, это дает воз-

можность “косвенного” доступа к этому объекту через указа-

тель. Предположим, что х - переменная, например, типа INT, а

рх - указатель, созданный неким еще не указанным способом.

Унарная операция & выдает адрес объекта, так что оператор

рх = &х;

присваивает адрес х переменной рх; говорят, что рх “ука-

зывает” на х. Операция & применима только к переменным и

элементам массива, конструкции вида &(х-1) и &3 являются не-

законными. Нельзя также получить адрес регистровой перемен-

ной.

Унарная операция \* рассматривает свой операнд как адрес

конечной цели и обращается по этому адресу, чтобы извлечь

содержимое. Следовательно, если Y тоже имеет тип INT, то

Y = \*рх;

присваивает Y содержимое того, на что указывает рх. Так пос-

ледовательность

рх = &х;

Y = \*рх;

присваивает Y то же самое значение, что и оператор

Y = X;

Переменные, участвующие во всем этом необходимо описать:

INT X, Y;

INT \*PX;

* 99 -

с описанием для X и Y мы уже неодонократно встречались.

Описание указателя

INT \*PX;

является новым и должно рассматриваться как мнемоническое;

оно говорит, что комбинация \*PX имеет тип INT. Это означает,

что если PX появляется в контексте \*PX, то это эквивалентно

переменной типа INT. Фактически синтаксис описания перемен-

ной имитирует синтаксис выражений, в которых эта переменная

может появляться. Это замечание полезно во всех случаях,

связанных со сложными описаниями. Например,

DOUBLE ATOF(), \*DP;

говорит, что ATOF() и \*DP имеют в выражениях значения типа

DOUBLE.

Вы должны также заметить, что из этого описания следу-

ет, что указатель может указывать только на определенный вид

объектов.

Указатели могут входить в выражения. Например, если PX

указывает на целое X, то \*PX может появляться в любом кон-

тексте, где может встретиться X. Так оператор

Y = \*PX + 1

присваивает Y значение, на 1 большее значения X;

PRINTF(“%D\N”, \*PX)

печатает текущее значение X;

D = SQRT((DOUBLE) \*PX)

получает в D квадратный корень из X, причем до передачи фун-

кции SQRT значение X преобразуется к типу DOUBLE. (Смотри

главу 2).

В выражениях вида

Y = \*PX + 1

унарные операции \* и & связаны со своим операндом более

крепко, чем арифметические операции, так что такое выражение

берет то значение, на которое указывает PX, прибавляет 1 и

присваивает результат переменной Y. Мы вскоре вернемся к то-

му, что может означать выражение

Y = \*(PX + 1)

Ссылки на указатели могут появляться и в левой части

присваиваний. Если PX указывает на X, то

\*PX = 0

* 100 -

полагает X равным нулю, а

\*PX += 1

увеличивает его на единицу, как и выражение

(\*PX)++

Круглые скобки в последнем примере необходимы; если их опус-

тить, то поскольку унарные операции, подобные \* и ++, выпол-

няются справа налево, это выражение увеличит PX, а не ту пе-

ременную, на которую он указывает.

И наконец, так как указатели являются переменными, то с

ними можно обращаться, как и с остальными переменными. Если

PY - другой указатель на переменную типа INT, то

PY = PX

копирует содержимое PX в PY, в результате чего PY указывает

на то же, что и PX.

5.2. Указатели и аргументы функций

Так как в “с” передача аргументов функциям осуществляет-

ся “по значению”, вызванная процедура не имеет непосредст-

венной возможности изменить переменную из вызывающей прог-

раммы. Что же делать, если вам действительно надо изменить

аргумент? например, программа сортировки захотела бы поме-

нять два нарушающих порядок элемента с помощью функции с

именем SWAP. Для этого недостаточно написать

SWAP(A, B);

определив функцию SWAP при этом следующим образом:

SWAP(X, Y) /\* WRONG \*/

INT X, Y;

{

INT TEMP;

TEMP = X;

X = Y;

Y = TEMP;

}

из-за вызова по значению SWAP не может воздействовать на

агументы A и B в вызывающей функции.

К счастью, все же имеется возможность получить желаемый

эффект. Вызывающая программа передает указатели подлежащих

изменению значений:

SWAP(&A, &B);

* 101 -

так как операция & выдает адрес переменной, то &A является

указателем на A. В самой SWAP аргументы описываются как ука-

затели и доступ к фактическим операндам осуществляется через

них.

SWAP(PX, PY) /\* INTERCHANGE \*PX AND \*PY \*/

INT \*PX, \*PY;

{

INT TEMP;

TEMP = \*PX;

\*PX = \*PY;

\*PY = TEMP;

}

Указатели в качестве аргументов обычно используются в

функциях, которые должны возвращать более одного значения.

(Можно сказать, что SWAP вOзвращает два значения, новые зна-

чения ее аргументов). В качестве примера рассмотрим функцию

GETINT, которая осуществляет преобразование поступающих в

своболном формате данных, разделяя поток символов на целые

значения, по одному целому за одно обращение. Функция GETINT

должна возвращать либо найденное значение, либо признак кон-

ца файла, если входные данные полностью исчерпаны. Эти зна-

чения должны возвращаться как отдельные объекты, какое бы

значение ни использовалось для EOF, даже если это значение

вводимого целого.

Одно из решений, основывающееся на описываемой в главе 7

функции ввода SCANF, состоит в том, чтобы при выходе на ко-

нец файла GETINT возвращала EOF в качестве значения функции;

любое другое возвращенное значение говорит о нахождении нор-

мального целого. Численное же значение найденного целого

возвращается через аргумент, который должен быть указателем

целого. Эта организация разделяет статус конца файла и чис-

ленные значения.

Следующий цикл заполняет массив целыми с помощью обраще-

ний к функции GETINT:

INT N, V, ARRAY[SIZE];

FOR (N = 0; N < SIZE && GETINT(&V) != EOF; N++)

ARRAY[N] = V;

В результате каждого обращения V становится равным следующе-

му целому значению, найденному во входных данных. Обратите

внимание, что в качестве аргумента GETINT необходимо указать

&V а не V. Использование просто V скорее всего приведет к

ошибке адресации, поскольку GETINT полагает, что она работа-

ет именно с указателем.

* 102 -

Сама GETINT является очевидной модификацией написанной

нами ранее функции ATOI:

GETINT(PN) /\* GET NEXT INTEGER FROM INPUT \*/

INT \*PN;

{

INT C,SIGN;

WHILE ((C = GETCH()) == ' ' \!\! C == '\N' \!\! C == '\T'); /\* SKIP WHITE SPACE \*/ SIGN = 1;

IF (C == '+' \!\! C == '-') { /\* RECORD

SIGN \*/

SIGN = (C == '+') ? 1 : -1;

C = GETCH();

}

FOR (\*PN = 0; C >= '0' && C <= '9'; C = GETCH())

\*PN = 10 \* \*PN + C - '0';

\*PN \*= SIGN;

IF (C != EOF)

UNGETCH©;

RETURN©;

}

Выражение \*PN используется всюду в GETINT как обычная пере-

менная типа INT. Мы также использовали функции GETCH и

UNGETCH (описанные в главе 4) , так что один лишний символ,

кототрый приходится считывать, может быть помещен обратно во

ввод.

Упражнение 5-1.

Напишите функцию GETFLOAT, аналог GETINT для чисел с

плавающей точкой. Какой тип должна возвращать GETFLOAT в ка-

честве значения функции?

5.3. Указатели и массивы

В языке “C” существует сильная взаимосвязь между указа-

телями и массивами , настолько сильная, что указатели и мас-

сивы действительно следует рассматривать одновременно. Любую

операцию, которую можно выполнить с помощью индексов масси-

ва, можно сделать и с помощью указателей. вариант с указате-

лями обычно оказывается более быстрым, но и несколько более

трудным для непосредственного понимания, по крайней мере для

начинающего. описание

INT A[10]

определяет массив размера 10, т.е. Набор из 10 последова-

тельных объектов, называемых A[0], A[1], ..., A[9]. Запись

A[I] соответствует элементу массива через I позиций от нача-

ла. Если PA - указатель целого, описанный как

* 103 -

INT \*PA

то присваивание

PA = &A[0]

приводит к тому, что PA указывает на нулевой элемент массива

A; это означает, что PA содержит адрес элемента A[0]. Теперь

присваивание

X = \*PA

будет копировать содержимое A[0] в X.

Если PA указывает на некоторый определенный элемент мас-

сива A, то по определению PA+1 указывает на следующий эле-

мент, и вообще PA-I указывает на элемент, стоящий на I пози-

ций до элемента, указываемого PA, а PA+I на элемент, стоящий

на I позиций после. Таким образом, если PA указывает на

A[0], то

\*(PA+1)

ссылается на содержимое A[1], PA+I - адрес A[I], а \*(PA+I) -

содержимое A[I].

Эти замечания справедливы независимо от типа переменных

в массиве A. Суть определения “добавления 1 к указателю”, а

также его распространения на всю арифметику указателей, сос-

тоит в том, что приращение масштабируется размером памяти,

занимаемой объектом, на который указывает указатель. Таким

образом, I в PA+I перед прибавлением умножается на размер

объектов, на которые указывает PA.

Очевидно существует очень тесное соответствие между ин-

дексацией и арифметикой указателей. в действительности ком-

пилятор преобразует ссылку на массив в указатель на начало

массива. В результате этого имя массива является указатель-

ным выражением. Отсюда вытекает несколько весьма полезных

следствий. Так как имя массива является синонимом местополо-

жения его нулевого элемента, то присваивание PA=&A[0] можно

записать как

PA = A

Еще более удивительным, по крайней мере на первый взг-

ляд, кажется тот факт, что ссылку на A[I] можно записать в

виде \*(A+I). При анализировании выражения A[I] в языке “C”

оно немедленно преобразуется к виду \*(A+I); эти две формы

совершенно эквивалентны. Если применить операцию & к обеим

частям такого соотношения эквивалентности, то мы получим,

что &A[I] и A+I тоже идентичны: A+I - адрес I-го элемента от

начала A. С другой стороны, если PA является указателем, то

в выражениях его можно использовать с индексом: PA[I] иден-

тично \*(PA+I). Короче, любое выражение, включающее массивы и

индексы, может быть записано через указатели и смещения и

наоборот, причем даже в одном и том же утверждении.

* 104 -

Имеется одно различие между именем массива и указателем,

которое необходимо иметь в виду. указатель является перемен-

ной, так что операции PA=A и PA++ имеют смысл. Но имя масси-

ва является константой, а не переменной: конструкции типа

A=PA или A++,или P=&A будут незаконными.

Когда имя массива передается функции, то на самом деле

ей передается местоположение начала этого массива. Внутри

вызванной функции такой аргумент является точно такой же пе-

ременной, как и любая другая, так что имя массива в качестве

аргумента действительно является указателем, т.е. Перемен-

ной, содержащей адрес. мы можем использовать это обстоятель-

ство для написания нового варианта функции STRLEN, вычисляю-

щей длину строки.

STRLEN(S) /\* RETURN LENGTH OF STRING S \*/

CHAR \*S;

{

INT N;

FOR (N = 0; \*S != '\0'; S++)

N++;

RETURN(N);

}

Операция увеличения S совершенно законна, поскольку эта

переменная является указателем; S++ никак не влияет на сим-

вольную строку в обратившейся к STRLEN функции, а только

увеличивает локальную для функции STRLEN копию адреса. Опи-

сания формальных параметров в определении функции в виде

CHAR S[];

CHAR \*S;

совершенно эквивалентны; какой вид описания следует предпо-

честь, определяется в значительной степени тем, какие выра-

жения будут использованы при написании функции. Если функции

передается имя массива, то в зависимости от того, что удоб-

нее, можно полагать, что функция оперирует либо с массивом,

либо с указателем, и действовать далее соответвующим обра-

зом. Можно даже использовать оба вида операций, если это ка-

жется уместным и ясным.

Можно передать функции часть массива, если задать в ка-

честве аргумента указатель начала подмассива. Например, если

A - массив, то как

F(&A[2])

как и

F(A+2)

* 105 -

передают функции F адрес элемента A[2], потому что и &A[2],

и A+2 являются указательными выражениями, ссылающимися на

третий элемент A. внутри функции F описания аргументов могут

присутствовать в виде:

F(ARR)

INT ARR[];

{

...

}

или

F(ARR)

INT \*ARR;

{

...

}

Что касается функции F, то тот факт, что ее аргумент в дейс-

твительности ссылается к части большего массива,не имеет для

нее никаких последствий.

5.4. Адресная арифметика

Если P является указателем, то каков бы ни был сорт

объекта, на который он указывает, операция P++ увеличивает P

так, что он указывает на следующий элемент набора этих

объектов, а операция P +=I увеличивает P так, чтобы он ука-

зывал на элемент, отстоящий на I элементов от текущего эле-

мента.эти и аналогичные конструкции представляют собой самые

простые и самые распространенные формы арифметики указателей

или адресной арифметики.

Язык “C” последователен и постоянен в своем подходе к

адресной арифметике; объединение в одно целое указателей,

массивов и адресной арифметики является одной из наиболее

сильных сторон языка. Давайте проиллюстрируем некоторые из

соответствующих возможностей языка на примере элементарной

(но полезной, несмотря на свою простоту) программы распреде-

ления памяти. Имеются две функции: функция ALLOC(N) возвра-

щает в качестве своего значения указатель P, который указы-

вает на первую из N последовательных символьных позиций, ко-

торые могут быть использованы вызывающей функцию ALLOC прог-

раммой для хранения символов; функция FREE(P) освобождает

приобретенную таким образом память, так что ее в дальнейшем

можно снова использовать. программа является “элементарной”,

потому что обращения к FREE должны производиться в порядке,

обратном тому, в котором производились обращения к ALLOC.

Таким образом, управляемая функциями ALLOC и FREE память яв-

ляется стеком или списком, в котором последний вводимый эле-

мент извлекается первым. Стандартная библиотека языка “C”

содержит аналогичные функции, не имеющие таких ограничений,

* 106 -

и, кроме того, в главе 8 мы приведем улучшенные варианты.

Между тем, однако, для многих приложений нужна только триви-

альная функция ALLOC для распределения небольших участков

памяти неизвестных заранее размеров в непредсказуемые момен-

ты времени.

Простейшая реализация состоит в том, чтобы функция раз-

давала отрезки большого символьного массива, которому мы

присвоили имя ALLOCBUF. Этот массив является собственностью

функций ALLOC и FREE. Так как они работают с указателями, а

не с индексами массива, никакой другой функции не нужно

знать имя этого массива. Он может быть описан как внешний

статический, т.е. Он будет локальным по отношению к исходно-

му файлу, содержащему ALLOC и FREE, и невидимым за его пре-

делами. При практической реализации этот массив может даже

не иметь имени; вместо этого он может быть получен в резуль-

тате запроса к операционной системе на указатель некоторого

неименованного блока памяти.

Другой необходимой информацией является то, какая часть

массива ALLOCBUF уже использована. Мы пользуемся указателем

первого свободного элемента, названным ALLOCP. Когда к функ-

ции ALLOC обращаются за выделением N символов, то она прове-

ряет, достаточно ли осталось для этого места в ALLOCBUF. Ес-

ли достаточно, то ALLOC возвращает текущее значение ALLOCP

(т.е. Начало свободного блока), затем увеличивает его на N,

с тем чтобы он указывал на следующую свободную область. Фун-

кция FREE(P) просто полагает ALLOCP равным P при условии,

что P указывает на позицию внутри ALLOCBUF.

DEFINE NULL 0 /\* POINTER VALUE FOR ERROR REPORT \*/

DEFINE ALLOCSIZE 1000 /\* SIZE OF AVAILABLE SPACE \*/

TATIC CHAR ALLOCBUF[ALLOCSIZE];/\* STORAGE FOR ALLOC \*/

TATIC CHAR **ALLOCP = ALLOCBUF; /** NEXT FREE POSITION \*/

HAR **ALLOC(N) /** RETURN POINTER TO N CHARACTERS \*/ INT N;

(

IF (ALLOCP + N <= ALLOCBUF + ALLOCSIZE) {

ALLOCP += N;

RETURN(ALLOCP - N); /\* OLD P \*/

} ELSE /\* NOT ENOUGH ROOM \*/

RETURN(NULL);

)

REE(P) /\* FREE STORAGE POINTED BY P \*/

HAR \*P;

(

IF (P >= ALLOCBUF && P < ALLOCBUF + ALLOCSIZE)

ALLOCP = P;

)

* 107 -

Дадим некоторые пояснения. Вообще говоря, указатель мо-

жет быть инициализирован точно так же, как и любая другая

переменная, хотя обычно единственными осмысленными значения-

ми являются NULL (это обсуждается ниже) или выражение, вклю-

чающее адреса ранее определенных данных соответствующего ти-

па. Описание

STATIC CHAR \*ALLOCP = ALLOCBUF;

определяет ALLOCP как указатель на символы и инициализирует

его так, чтобы он указывал на ALLOCBUF, т.е. На первую сво-

бодную позицию при начале работы программы. Так как имя мас-

сива является адресом его нулевого элемента, то это можно

было бы записать в виде

STATIC CHAR \*ALLOCP = &ALLOCBUF[0];

используйте ту запись, которая вам кажется более естествен-

ной. С помощью проверки

IF (ALLOCP + N <= ALLOCBUF + ALLOCSIZE)

выясняется, осталось ли достаточно места, чтобы удовлетво-

рить запрос на N символов. Если достаточно, то новое значе-

ние ALLOCP не будет указывать дальше, чем на последнюю пози-

цию ALLOCBUF. Если запрос может быть удовлетворен, то ALLOC

возвращает обычный указатель (обратите внимание на описание

самой функции). Если же нет, то ALLOC должна вернуть некото-

рый признак, говорящий о том, что больше места не осталось.

В языке “C” гарантируется, что ни один правильный указатель

данных не может иметь значение нуль, так что возвращение ну-

ля может служить в качестве сигнала о ненормальном событии,

в данном случае об отсутствии места. Мы, однако, вместо нуля

пишем NULL, с тем чтобы более ясно показать, что это специ-

альное значение указателя. Вообще говоря, целые не могут ос-

мысленно присваиваться указателям, а нуль - это особый слу-

чай.

Проверки вида

IF (ALLOCP + N <= ALLOCBUF + ALOOCSIZE)

и

IF (P >= ALLOCBUF && P < ALLOCBUF + ALLOCSIZE)

демонстрируют несколько важных аспектов арифметики указате-

лей. Во-первых , при определенных условиях указатели можно

сравнивать. Если P и Q указывают на элементы одного и того

же массива, то такие отношения, как <, >= и т.д., работают

надлежащим образом. Например,

P < Q

* 108 -

истинно, если P указывает на более ранний элемент массива,

чем Q. Отношения == и != тоже работают. Любой указатель мож-

но осмысленным образом сравнить на равенство или неравенство

с NULL. Но ни за что нельзя ручаться, если вы используете

сравнения при работе с указателями, указывающими на разные

массивы. Если вам повезет, то на всех машинах вы получите

очевидную бессмыслицу. Если же нет, то ваша программа будет

правильно работать на одной машине и давать непостижимые ре-

зультаты на другой.

Во-вторых, как мы уже видели, указатель и целое можно

складывать и вычитать. Конструкция

P + N

подразумевает N-ый объект за тем, на который P указывает в

настоящий момент. Это справедливо независимо от того, на ка-

кой вид объектов P должен указывать; компилятор сам масшта-

бирует N в соответствии с определяемым из описания P разме-

ром объектов, указываемых с помощью P. например, на PDP-11

масштабирующий множитель равен 1 для CHAR, 2 для INT и

SHORT, 4 для LONG и FLOAT и 8 для DOUBLE.

Вычитание указателей тоже возможно: если P и Q указывают

на элементы одного и того же массива, то P-Q - количество

элементов между P и Q. Этот факт можно использовать для на-

писания еще одного варианта функции

STRLEN:

STRLEN(S) /\* RETURN LENGTH OF STRING S \*/

CHAR \*S;

{

CHAR \*P = S;

WHILE (\*P != '\0')

P++;

RETURN(P-S);

}

При описании указатель P в этой функции инициализирован

посредством строки S, в результате чего он указывает на пер-

вый символ строки. В цикле WHILE по очереди проверяется каж-

дый символ до тех пор, пока не появится символ конца строки

\0. Так как значение \0 равно нулю, а WHILE только выясняет,

имеет ли выражение в нем значение 0, то в данном случае яв-

ную проверку можно опустить. Такие циклы часто записывают в

виде

WHILE (\*P)

P++;

Так как P указывает на символы, то оператор P++ передви-

гает P каждый раз так, чтобы он указывал на следующий сим-

вол. В результате P-S дает число просмотренных символов,

* 108 -

т.е. Длину строки. Арифметика указателей последовательна:

если бы мы имели дело с переменными типа FLOAT, которые за-

нимают больше памяти, чем переменные типа CHAR, и если бы P

был указателем на FLOAT, то оператор P++ передвинул бы P на

следующее FLOAT. таким образом, мы могли бы написать другой

вариант функции ALLOC, распределяющей память для FLOAT,

вместо CHAR, просто заменив всюду в ALLOC и FREE описатель

CHAR на FLOAT. Все действия с указателями автоматически учи-

тывают размер объектов, на которые они указывают, так что

больше ничего менять не надо.

За исключением упомянутых выше операций (сложение и вы-

читание указателя и целого, вычитание и сравнение двух ука-

зателей), вся остальная арифметика указателей является неза-

конной. Запрещено складывать два указателя, умножать, де-

лить, сдвигать или маскировать их, а также прибавлять к ним

переменные типа FLOAT или DOUBLE.

5.5. Указатели символов и функции

Строчная константа, как, например,

“I AM A STRING”

является массивом символов. Компилятор завершает внутреннее

представление такого массива символом \0, так что программы

могут находить его конец. Таким образом, длина массива в па-

мяти оказывается на единицу больше числа символов между

двойными кавычками.

По-видимому чаще всего строчные константы появляются в

качестве аргументов функций, как, например, в

PRINTF (“HELLO, WORLD\N”);

когда символьная строка, подобная этой, появляется в прог-

рамме, то доступ к ней осуществляется с помощью указателя

символов; функция PRINTF фактически получает указатель сим-

вольного массива.

Конечно, символьные массивы не обязаны быть только аргу-

ментами функций. Если описать MESSAGE как

CHAR \*MESSAGE;

то в результате оператора

MESSAGE = “NOW IS THE TIME”;

переменная MESSAGE станет указателем на фактический массив

символов. Это не копирование строки; здесь участвуют только

указатели. в языке “C” не предусмотрены какие-либо операции

для обработки всей строки символов как целого.

Мы проиллюстрируем другие аспекты указателей и массивов,

разбирая две полезные функции из стандартной библиотеки вво-

да-вывода, которая будет рассмотрена в главе 7.

* 109 -

Первая функция - это STRCPY(S,T), которая копирует стро-

ку т в строку S. Аргументы написаны именно в этом порядке по

аналогии с операцией присваивания, когда для того, чтобы

присвоить T к S обычно пишут

S = T

сначала приведем версию с массивами:

STRCPY(S, T) /\* COPY T TO S \*/

CHAR S[], T[];

{

INT I;

I = 0;

WHILE ((S[I] = T[I]) != '\0')

I++;

}

Для сопоставления ниже дается вариант STRCPY с указате-

лями.

STRCPY(S, T) /\* COPY T TO S; POINTER VERSION 1 \*/ CHAR \*S, \*T;

{

WHILE ((\*S = \*T) != '\0') {

S++;

T++;

}

}

Так как аргументы передаются по значению, функция STRCPY

может использовать S и T так, как она пожелает. Здесь они с

удобством полагаются указателями, которые передвигаются

вдоль массивов, по одному символу за шаг, пока не будет ско-

пирован в S завершающий в T символ \0.

На практике функция STRCPY была бы записана не так, как

мы показали выше. Вот вторая возможность:

STRCPY(S, T) /\* COPY T TO S; POINTER VERSION 2 \*/ CHAR \*S, \*T;

{

WHILE ((\*S++ = \*T++) != '\0')

;

}

Здесь увеличение S и T внесено в проверочную часть. Зна-

чением \*T++ является символ, на который указывал T до увели-

чения; постфиксная операция ++ не изменяет T, пока этот сим-

вол не будет извлечен. Точно так же этот символ помещается в

старую позицию S, до того как S будет увеличено. Конечный

результат заключается в том, что все символы, включая завер-

шающий \0, копируются из T в S.

* 110 -

И как последнее сокращение мы опять отметим, что сравне-

ние с \0 является излишним, так что функцию можно записать в

виде

STRCPY(S, T) /\* COPY T TO S; POINTER VERSION 3 \*/

CHAR \*S, \*T;

{

WHILE (\*S++ = \*T++)

;

}

хотя с первого взгляда эта запись может показаться загадоч-

ной, она дает значительное удобство. Этой идиомой следует

овладеть уже хотя бы потому, что вы с ней будете часто вст-

речаться в “C”-программах.

Вторая функция - STRCMP(S, T), которая сравнивает сим-

вольные строки S и т, возвращая отрицательное, нулевое или

положительное значение в соответствии с тем, меньше, равно

или больше лексикографически S, чем T. Возвращаемое значение

получается в результате вычитания символов из первой пози-

ции, в которой S и T не совпадают.

STRCMP(S, T) /\* RETURN <0 IF S<T, 0 IF S==T, >0 IF S>T \*/

CHAR S[], T[];

{

INT I;

I = 0;

WHILE (S[I] == T[I])

IF (S[I++] == '\0')

RETURN(0);

RETURN(S[I]-T[I]);

}

Вот версия STRCMP с указателями:

STRCMP(S, T) /\* RETURN <0 IF S<T, 0 IF S==T, >0 IF S>T \*/ CHAR \*S, \*T;

{

FOR ( ; \*S == \*T; S++, T++)

IF (\*S == '\0')

RETURN(0);

RETURN(\*S-\*T);

}

так как ++ и—могут быть как постфиксными, так и

префиксными операциями, встречаются другие комбинации \* и

++ и --, хотя и менее часто.

Например

\*++P

* 111 -

увеличивает P до извлечения символа, на который указывает

P, а

\*--P

сначала уменьшает P.

Упражнение 5-2.

Напишите вариант с указателями функции STRCAT из главы

2: STRCAT(S, T) копирует строку T в конец S.

Упражнение 5-3.

Напишите макрос для STRCPY.

Упражнение 5-4.

Перепишите подходящие программы из предыдущих глав и уп-

ражнений, используя указатели вместо индексации массивов.

Хорошие возможности для этого предоставляют функции GETLINE

/главы 1 и 4/, ATOI, ITOA и их варианты /главы 2, 3 и 4/,

REVERSE /глава 3/, INDEX и GETOP /глава 4/.

5.6. Указатели - не целые.

Вы, возможно, обратили внимание в предыдущих “с”-прог-

раммах на довольно непринужденное отношение к копированию

указателей. В общем это верно, что на большинстве машин ука-

затель можно присвоить целому и передать его обратно, не из-

менив его; при этом не происходит никакого масштабирования

или преобразования и ни один бит не теряется. к сожалению,

это ведет к вольному обращению с функциями, возвращающими

указатели, которые затем просто передаются другим функциям,

* необходимые описания указателей часто опускаются. Рассмот-

рим, например, функцию STRSAVE(S), которая копирует строку S

в некоторое место для хранения, выделяемое посредством обра-

щения к функции ALLOC, и возвращает указатель на это место.

Правильно она должна быть записана так:

CHAR **STRSAVE(S) /** SAVE STRING S SOMEWHERE \*/ CHAR \*S;

{

CHAR \*P, \*ALLOC();

IF ((P = ALLOC(STRLEN(S)+1)) != NULL)

STRCPY(P, S);

RETURN(P);

}

на практике существует сильное стремление опускать описания:

* 112 -

**STRSAVE(S) /** SAVE STRING S SOMEWHERE \*/

{

CHAR \*P;

IF ((P = ALLOC(STRLEN(S)+1)) != NULL)

STRCPY(P, S);

RETURN(P);

}

Эта программа будет правильно работать на многих маши-

нах, потому что по умолчанию функции и аргументы имеют тип

INT, а указатель и целое обычно можно безопасно пересылать

туда и обратно. Однако такой стиль программирования в своем

существе является рискованным, поскольку зависит от деталей

реализации и архитектуры машины и может привести к непра-

вильным результатам на конкретном используемом вами компиля-

торе. Разумнее всюду использовать полные описания. (Отладоч-

ная программа LINT предупредит о таких конструкциях, если

они по неосторожности все же появятся).

5.7. Многомерные массивы.

В языке “C” предусмотрены прямоугольные многомерные мас-

сивы, хотя на практике существует тенденция к их значительно

более редкому использованию по сравнению с массивами указа-

телей. В этом разделе мы рассмотрим некоторые их свойства.

Рассмотрим задачу преобразования дня месяца в день года

и наоборот. Например, 1-ое марта является 60-м днем невисо-

косного года и 61-м днем високосного года. Давайте введем

две функции для выполнения этих преобразований: DAY\_OF\_YEAR

преобразует месяц и день в день года, а MONTH\_DAY преобразу-

ет день года в месяц и день. Так как эта последняя функция

возвращает два значения, то аргументы месяца и дня должны

быть указателями:

MONTH\_DAY(1977, 60, &M, &D)

Полагает M равным 3 и D равным 1 (1-ое марта).

Обе эти функции нуждаются в одной и той же информацион-

ной таблице, указывающей число дней в каждом месяце. Так как

число дней в месяце в високосном и в невисокосном году отли-

чается, то проще представить их в виде двух строк двумерного

массива, чем пытаться прослеживать во время вычислений, что

именно происходит в феврале. Вот этот массив и выполняющие

эти преобразования функции:

STATIC INT DAY\_TAB[2][13] = {

(0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31),

(0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

};

* 113 -

DAY\_OF\_YEAR(YEAR, MONTH, DAY) /\* SET DAY OF YEAR \*/

INT YEAR, MONTH, DAY; /\* FROM MONTH & DAY \*/

{

INT I, LEAP;

LEAP = YEAR%4 == 0 && YEAR%100 != 0 \!\! YEAR%400 == 0;

FOR (I = 1; I < MONTH; I++)

DAY += DAY\_TAB[LEAP][I];

RETURN(DAY);

{

MONTH\_DAY(YEAR, YEARDAY, PMONTH, PDAY) /\*SET MONTH,DAY \*/

INT YEAR, YEARDAY, \*PMONTH, **PDAY; /** FROM DAY OF YEAR \*/

{

LEAP = YEAR%4 == 0 && YEAR%100 != 0 \!\! YEAR%400 == 0;

FOR (I = 1; YEARDAY > DAY\_TAB[LEAP][I]; I++)

YEARDAY -= DAY\_TAB[LEAP][I];

\*PMONTH = I;

\*PDAY = YEARDAY;

}

Массив DAY\_TAB должен быть внешним как для DAY\_OF\_YEAR, так

и для MONTH\_DAY, поскольку он используется обеими этими фун-

кциями.

Массив DAY\_TAB является первым двумерным массивом, с ко-

торым мы имеем дело. По определению в “C” двумерный массив

по существу является одномерным массивом, каждый элемент ко-

торого является массивом. Поэтому индексы записываются как

DAY\_TAB[I][J]

а не

DAY\_TAB [I, J]

как в большинстве языков. В остальном с двумерными массивами

можно в основном обращаться таким же образом, как в других

языках. Элементы хранятся по строкам, т.е. При обращении к

элементам в порядке их размещения в памяти быстрее всего из-

меняется самый правый индекс.

Массив инициализируется с помощью списка начальных зна-

чений, заключенных в фигурные скобки; каждая строка двумер-

ного массива инициализируется соответствующим подсписком. Мы

поместили в начало массива DAY\_TAB столбец из нулей для то-

го, чтобы номера месяцев изменялись естественным образом от

1 до 12, а не от 0 до 11. Так как за экономию памяти у нас пока не награждают, такой способ проще, чем подгонка индек-сов.

Если двумерный массив передается функции, то описание

соответствующего аргумента функции должно содержать количес-

тво столбцов; количество строк несущественно, поскольку, как

и прежде, фактически передается указатель. В нашем конкрет-

ном случае это указатель объектов, являющихся массивами из

* 114 -

13 чисел типа INT. Таким образом, если бы требовалось пере-

дать массив DAY\_TAB функции F, то описание в F имело бы вид:

F(DAY\_TAB)

INT DAY\_TAB[2][13];

{

...

}

Так как количество строк является несущественным, то описа-

ние аргумента в F могло бы быть таким:

INT DAY\_TAB[][13];

или таким

INT (\*DAY\_TAB)[13];

в которм говорится, что аргумент является указателем массива

из 13 целых. Круглые скобки здесь необходимы, потому что

квадратные скобки [] имеют более высокий уровень старшинст-

ва, чем \*; как мы увидим в следующем разделе, без круглых

скобок

INT \*DAY\_TAB[13];

является описанием массива из 13 указателей на целые.

5.8. Массивы указателей; указатели указателей

Так как указатели сами являются переменными, то вы впол-

не могли бы ожидать использования массива указателей. Это

действительно так. Мы проиллюстрируем это написанием прог-

раммы сортировки в алфавитном порядке набора текстовых

строк, предельно упрощенного варианта утилиты SORT операци-

онной систем UNIX.

В главе 3 мы привели функцию сортировки по шеллу, кото-

рая упорядочивала массив целых. Этот же алгоритм будет рабо-

тать и здесь, хотя теперь мы будем иметь дело со строчками

текста различной длины, которые, в отличие от целых, нельзя

сравнивать или перемещать с помощью одной операции. Мы нуж-

даемся в таком представлении данных, которое бы позволяло

удобно и эффективно обрабатывать строки текста переменной

длины.

Здесь и возникают массивы указателей. Если подлежащие

сортировке сроки хранятся одна за другой в длинном символь-

ном массиве (управляемом, например, функцией ALLOC), то к

каждой строке можно обратиться с помощью указателя на ее

первый символ. Сами указатели можно хранить в массиве. две

строки можно сравнить, передав их указатели функции STRCMP.

* 115 -

Если две расположенные в неправильном порядке строки должны

быть переставлены, то фактически переставляются указатели в

массиве указателей, а не сами тексты строк. Этим исключаются

сразу две связанные проблемы: сложного управления памятью и

больших дополнительных затрат на фактическую перестановку

строк.

Процесс сортировки включает три шага:

чтение всех строк ввода

их сортировка

вывод их в правильном порядке

Как обычно, лучше разделить программу на несколько функций в

соответствии с естественным делением задачи и выделить веду-

щую функцию, управляющую работой всей программы.

Давайте отложим на некоторое время рассмотрение шага сорти-

ровки и сосредоточимся на структуре данных и вводе-выводе.

Функция, осуществляющая ввод, должна извлечь символы каждой

строки, запомнить их и построить массив указателей строк.

Она должна также подсчитать число строк во вводе, так как

эта информация необходима при сортировке и выводе. так как

функция ввода в состоянии справиться только с конечным чис-

лом вводимых строк, в случае слишком большого их числа она

может возвращать некоторое число, отличное от возможного

числа строк, например -1. Функция осуществляющая вывод, дол-

жна печатать строки в том порядке, в каком они появляются в

массиве указателей.

#DEFINE NULL 0

#DEFINE LINES 100 /\* MAX LINES TO BE SORTED \*/

MAIN() /\* SORT INPUT LINES \*/

\(

CHAR \*LINEPTR[LINES]; /\*POINTERS TO TEXT LINES \*/

INT NLINES; /\* NUMBER OF INPUT LINES READ \*/

IF ((NLINES = READLINES(LINEPTR, LINES)) >= 0) \( SORT(LINEPTR, NLINES);

WRITELINES(LINEPTR, NLINES);

\)

ELSE

PRINTF(“INPUT TOO BIG TO SORT\N”);

\)

#DEFINE MAXLEN 1000

* 116 -

READLINES(LINEPTR, MAXLINES) /\* READ INPUT LINES \*/

CHAR **LINEPTR[]; /** FOR SORTING \*/

INT MAXLINES;

\(

INT LEN, NLINES;

CHAR \*P, \*ALLOC(), LINE[MAXLEN];

NLINES = 0;

WHILE ((LEN = GETLINE(LINE, MAXLEN)) > 0)

IF (NLINES >= MAXLINES)

RETURN(-1);

ELSE IF ((P = ALLOC(LEN)) == NULL)

RETURN (-1);

ELSE \(

LINE[LEN-1] = '\0'; /\* ZAP NEWLINE \*/

STRCPY(P,LINE);

LINEPTR[NLINES++] = P;

\)

RETURN(NLINES);

\)

Символ новой строки в конце каждой строки удаляется, так что

он никак не будет влиять на порядок, в котором сортируются

строки.

WRITELINES(LINEPTR, NLINES) /\* WRITE OUTPUT LINES \*/

CHAR \*LINEPTR[];

INT NLINES;

\(

INT I;

FOR (I = 0; I < NLINES; I++)

PRINTF(“%S\N”, LINEPTR[I]);

\)

Существенно новым в этой программе является описание

CHAR \*LINEPTR[LINES];

которое сообщает, что LINEPTR является массивом из LINES

элементов, каждый из которых - указатель на переменные типа

CHAR. Это означает, что LINEPTR[I] - указатель на символы, а

\*LINEPTR[I] извлекает символ.

Так как сам LINEPTR является массивом, который передает-

ся функции WRITELINES, с ним можно обращаться как с указате-

лем точно таким же образом, как в наших более ранних приме-

* 117 -

рах. Тогда последнюю функцию можно переписать в виде:

WRITELINES(LINEPTR, NLINES) /\* WRITE OUTPUT LINES \*/ CHAR \*LINEPTR[];

INT NLINES;

\(

INT I;

WHILE (--NLINES >= 0)

PRINTF(“%S\N”, \*LINEPTR++);

\)

здесь \*LINEPTR сначала указывает на первую строку; каждое

увеличение передвигает указатель на следующую строку, в то

время как NLINES убывает до нуля.

Справившись с вводом и выводом, мы можем перейти к сор-

тировке. программа сортировки по шеллу из главы 3 требует

очень небольших изменений: должны быть модифицированы описа-

ния, а операция сравнения выделена в отдельную функцию. Ос-

новной алгоритм остается тем же самым, и это дает нам опре-

деленную уверенность, что он по-прежнему будет работать.

SORT(V, N) /\* SORT STRINGS V[0] ... V[N-1] \*/

CHAR **V[]; /** INTO INCREASING ORDER \*/

INT N;

\(

INT GAP, I, J;

CHAR \*TEMP;

FOR (GAP = N/2; GAP > 0; GAP /= 2)

FOR (I = GAP; I < N; I++)

FOR (J = I - GAP; J >= 0; J -= GAP) \(

IF (STRCMP(V[J], V[J+GAP]) <= 0)

BREAK;

TEMP = V[J];

V[J] = V[J+GAP];

V[J+GAP] = TEMP;

\)

\)

Так как каждый отдельный элемент массива V (имя формального

параметра, соответствующего LINEPTR) является указателем на

символы, то и TEMP должен быть указателем на символы, чтобы

их было можно копировать друг в друга.

Мы написали эту программу по возможности более просто с

тем, чтобы побыстрее получить работающую программу. Она мог-

ла бы работать быстрее, если, например, вводить строки не-

посредственно в массив, управляемый функцией READLINES, а не

копировать их в LINE, а затем в скрытое место с помощью фун-

* 118 -

кции ALLOC. но мы считаем, что будет разумнее первоначальный

вариант сделать более простым для понимания, а об “эффектив-

ности” позаботиться позднее. Все же, по-видимому, способ,

позволяющий добиться заметного ускорения работы программы

состоит не в исключении лишнего копирования вводимых строк.

Более вероятно, что существенной разницы можно достичь за

счет замены сортировки по шеллу на нечто лучшее, например,

на метод быстрой сортировки.

В главе 1 мы отмечали, что поскольку в циклах WHILE и

FOR проверка осуществляется до того, как тело цикла выпол-

нится хотя бы один раз, эти циклы оказываются удобными для

обеспечения правильной работы программы при граничных значе-

ниях, в частности, когда ввода вообще нет. Очень полезно

просмотреть все функции программы сортировки, разбираясь,

что происходит, если вводимый текст отсутствует.

Упражнение 5-5.

Перепишите функцию READLINES таким образом, чтобы она

помещала строки в массив, предоставляемый функцией MAIN, а

не в память, управляемую обращениями к функции ALLOC. На-

сколько быстрее стала программа?

5.9. Инициализация массивов указателей.

Рассмотрим задачу написания функции MONTH\_NAME(N), кото-

рая возвращает указатель на символьную строку, содержащую

имя N-го месяца. Это идеальная задача для применения внут-

реннего статического массива. Функция MONTH\_NAME содержит

локальный массив символьных строк и при обращении к ней воз-

вращает указатель нужной строки. Тема настоящего раздела -

как инициализировать этот массив имен.

CHAR **MONTH\_NAME(N) /** RETURN NAME OF N-TH MONTH \*/

INT N;

\(

STATIC CHAR \*NAME[] = \(

“ILLEGAL MONTH”,

“JANUARY”,

“FEBRUARY”,

“MARCH”,

“APRIL”,

“MAY”,

“JUN”,

“JULY”,

“AUGUST”,

“SEPTEMBER”,

“OCTOBER”,

“NOVEMBER”,

“DECEMBER”

\);

RETURN ((N < 1 \!\! N > 12) ? NAME[0] : NAME[N]);

\)

* 119 -

Описание массива указателей на символы NAME точно такое же,

как аналогичное описание LINEPTR в примере с сортировкой.

Инициализатором является просто список символьных строк;

каждая строка присваивается соответствующей позиции в масси-

ве. Более точно, символы I-ой строки помещаются в какое-то

иное место, а ее указатель хранится в NAME[I]. Поскольку

размер массива NAME не указан, компилятор сам подсчитывает

количество инициализаторов и соответственно устанавливает

правильное число.

5.10. Указатели и многомерные массивы

Начинающие изучать язык “с” иногда становятся в тупик

перед вопросом о различии между двумерным массивом и масси-

вом указателей, таким как NAME в приведенном выше примере.

Если имеются описания

INT A[10][10];

INT \*B[10];

то A и B можно использовать сходным образом в том смысле,

что как A[5][5], так и B[5][5] являются законными ссылками

на отдельное число типа INT. Но A - настоящий массив: под

него отводится 100 ячеек памяти и для нахождения любого ука-

занного элемента проводятся обычные вычисления с прямоуголь-

ными индексами. Для B, однако, описание выделяет только 10

указателей; каждый указатель должен быть установлен так,

чтобы он указывал на массив целых. если предположить, что

каждый из них указывает на массив из 10 элементов, то тогда

где-то будет отведено 100 ячеек памяти плюс еще десять ячеек

для указателей. Таким образом, массив указателей использует

несколько больший объем памяти и может требовать наличие яв-

ного шага инициализации. Но при этом возникают два преиму-

щества: доступ к элементу осуществляется косвенно через ука-

затель, а не посредством умножения и сложения, и строки мас-

сива могут иметь различные длины. Это означает, что каждый

элемент B не должен обязательно указывать на вектор из 10

элементов; некоторые могут указывать на вектор из двух эле-

ментов, другие - из двадцати, а третьи могут вообще ни на

что не указывать.

Хотя мы вели это обсуждение в терминах целых, несомнен-

но, чаще всего массивы указателей используются так, как мы

продемонстрировали на функции MONTH\_NAME, - для хранения

символьных строк различной длины.

Упражнение 5-6.

Перепишите функции DAY\_OF\_YEAR и MONTH\_DAY, используя

вместо индексации указатели.

* 120 -

5.11. Командная строка аргументов

Системные средства, на которые опирается реализация язы-

ка “с”, позволяют передавать командную строку аргументов или

параметров начинающей выполняться программе. Когда функция

MAIN вызывается к исполнению, она вызывается с двумя аргу-

ментами. Первый аргумент (условно называемый ARGC) указывает

число аргументов в командной строке, с которыми происходит

обращение к программе; второй аргумент (ARGV) является ука-

зателем на массив символьных строк, содержащих эти аргумен-

ты, по одному в строке. Работа с такими строками - это обыч-

ное использование многоуровневых указателей.

Самую простую иллюстрацию этой возможности и необходимых

при этом описаний дает программа ECHO, которая просто печа-

тает в одну строку аргументы командной строки, разделяя их

пробелами. Таким образом, если дана команда

ECHO HELLO, WORLD

то выходом будет

HELLO, WORLD

по соглашению ARGV[0] является именем, по которому вызывает-

ся программа, так что ARGC по меньшей мере равен 1. В приве-

денном выше примере ARGC равен 3, а ARGV[0], ARGV[1] и

ARGV[2] равны соответственно “ECHO”, “HELLO,” и “WORLD”.

Первым фактическим агументом является ARGV[1], а последним -

ARGV[ARGC-1]. Если ARGC равен 1, то за именем программы не

следует никакой командной строки аргументов. Все это показа-

но в ECHO:

MAIN(ARGC, ARGV) /\* ECHO ARGUMENTS; 1ST VERSION \*/

INT ARGC;

CHAR \*ARGV[];

\(

INT I;

FOR (I = 1; I < ARGC; I++)

PRINTF(“%S%C”, ARGV[I], (I<ARGC-1) ? ' ' : '\N');

\)

Поскольку ARGV является указателем на массив указателей, то

существует несколько способов написания этой программы, ис-

пользующих работу с указателем, а не с индексацией массива.

Мы продемонстрируем два варианта.

MAIN(ARGC, ARGV) /\* ECHO ARGUMENTS; 2ND VERSION \*/

INT ARGC;

CHAR \*ARGV[];

\(

WHILE (--ARGC > 0)

PRINTF(“%S%C”,\*++ARGV, (ARGC > 1) ? ' ' : '\N');

\)

* 121 -

Так как ARGV является указателем на начало массива строк-ар-

гументов, то, увеличив его на 1 (++ARGV), мы вынуждаем его

указывать на подлинный аргумент ARGV[1], а не на ARGV[0].

Каждое последующее увеличение передвигает его на следующий

аргумент; при этом \*ARGV становится указателем на этот аргу-

мент. одновременно величина ARGC уменьшается; когда она об-

ратится в нуль, все аргументы будут уже напечатаны.

Другой вариант:

MAIN(ARGC, ARGV) /\* ECHO ARGUMENTS; 3RD VERSION \*/ INT ARGC;

CHAR \*ARGV[];

\(

WHILE (--ARGC > 0)

PRINTF((ARGC > 1) ? “%S” : “%S\N”, \*++ARGV);

\)

Эта версия показывает, что аргумент формата функции PRINTF

может быть выражением, точно так же, как и любой другой. Та-

кое использование встречается не очень часто, но его все же

стоит запомнить.

Как второй пример, давайте внесем некоторые усовершенст-

вования в программу отыскания заданной комбинации символов

из главы 4. Если вы помните, мы поместили искомую комбинацию

глубоко внутрь программы, что очевидно является совершенно

неудовлетворительным. Следуя утилите GREP системы UNIX, да-

вайте изменим программу так, чтобы эта комбинация указыва-

лась в качестве первого аргумента строки.

#DEFINE MAXLINE 1000

MAIN(ARGC, ARGV) /\* FIND PATTERN FROM FIRST ARGUMENT \*/ INT ARGC;

CHAR \*ARGV[];

\(

CHAR LINE[MAXLINE];

IF (ARGC != 2)

PRINTF (“USAGE: FIND PATTERN\N”);

ELSE

WHILE (GETLINE(LINE, MAXLINE) > 0)

IF (INDEX(LINE, ARGV[1] >= 0)

PRINTF(“%S”, LINE);

\)

Теперь может быть развита основная модель, иллюстрирую-

щая дальнейшее использование указателей. Предположим, что

нам надо предусмотреть два необязательных аргумента. Один

утверждает: “напечатать все строки за исключением тех, кото-

рые содержат данную комбинацию”, второй гласит: “перед каж-

дой выводимой строкой должен печататься ее номер”.

* 122 -

Общепринятым соглашением в “с”-программах является то,

что аргумент, начинающийся со знака минус, вводит необяза-

тельный признак или параметр. Если мы, для того, чтобы сооб-

щить об инверсии, выберем -X, а для указания о нумерации

нужных строк выберем -N(“номер”), то команда

FIND -X -N THE

при входных данных

NOW IS THE TIME

FOR ALL GOOD MEN

TO COME TO THE AID

OF THEIR PARTY.

Должна выдать

2:FOR ALL GOOD MEN

Нужно, чтобы необязательные аргументы могли располагать-

ся в произвольном порядке, и чтобы остальная часть программы

не зависела от количества фактически присутствующих аргумен-

тов. в частности, вызов функции INDEX не должен содержать

ссылку на ARGV[2], когда присутствует один необязательный

аргумент, и на ARGV[1], когда его нет. Более того, для поль-

зователей удобно, чтобы необязательные аргументы можно было

объединить в виде:

FIND -NX THE

вот сама программа:

#DEFINE MAXLINE 1000

MAIN(ARGC, ARGV) /\* FIND PATTERN FROM FIRST ARGUMENT \*/ INT ARGC;

CHAR \*ARGV[];

\(

CHAR LINE[MAXLINE], \*S;

LONG LINENO = 0;

INT EXCEPT = 0, NUMBER = 0;

WHILE (--ARGC > 0 && (\*++ARGV)[0] == '-')

FOR (S = ARGV[0]+1; \*S != '\0'; S++)

SWITCH (\*S) \(

CASE 'X':

EXCEPT = 1;

BREAK;

* 123 -

CASE 'N':

NUMBER = 1;

BREAK;

DEFAULT:

PRINTF(“FIND: ILLEGAL OPTION %C\N”, \*S);

ARGC = 0;

BREAK;

\)

IF (ARGC != 1)

PRINTF(“USAGE: FIND -X -N PATTERN\N”);

ELSE

WHILE (GETLINе(LINE, MAXLINE) > 0) \(

LINENO++;

IF ((INDEX(LINE, \*ARGV) >= 0) != EXCEPT) \

IF (NUMBER)

PRINTF(“%LD: “, LINENO);

PRINTF(“%S”, LINE);

\)

\)

\)

Аргумент ARGV увеличивается перед каждым необязательным

аргументом, в то время как аргумент ARGC уменьшается. если

нет ошибок, то в конце цикла величина ARGC должна равняться

1, а \*ARGV должно указывать на заданную комбинацию. Обратите

внимание на то, что \*++ARGV является указателем аргументной

строки; (\*++ARGV)[0] - ее первый символ. Круглые скобки

здесь необходимы, потому что без них выражение бы приняло

совершенно отличный (и неправильный) вид \*++(ARGV[0]). Дру-

гой правильной формой была бы \*\*++ARGV.

Упражнение 5-7.

Напишите программу ADD, вычисляющую обратное польское

выражение из командной строки. Например,

ADD 2 3 4 + \*

вычисляет 2\*(3+4).

Упражнение 5-8.

Модифицируйте программы ENTAB и DETAB (указанные в ка-

честве упражнений в главе 1) так, чтобы они получали список

табуляционных остановок в качестве аргументов. Если аргумен-

ты отсутствуют, используйте стандартную установку табуляций.

Упражнение 5-9.

Расширьте ENTAB и DETAB таким образом, чтобы они воспри-

нимали сокращенную нотацию

ENTAB M +N

* 124 -

означающую табуляционные остановки через каждые N столбцов,

начиная со столбца M. Выберите удобное (для пользователя)

поведение функции по умолчанию.

Упражнение 5-10.

Напишите программу для функции TAIL, печатающей послед-

ние N строк из своего файла ввода. Пусть по умолчанию N рав-

но 10, но это число может быть изменено с помощью необяза-

тельного аргумента, так что

TAIL -N

печатает последние N строк. программа должна действовать ра-

ционально, какими бы неразумными ни были бы ввод или значе-

ние N. Составьте программу так, чтобы она оптимальным обра-

зом использовала доступную память: строки должны храниться,

как в функции SORT, а не в двумерном массиве фиксированного

размера.

5.12. Указатели на функции

В языке “с” сами функции не являются переменными, но

имеется возможность определить указатель на функцию, который

можно обрабатывать, передавать другим функциям, помещать в

массивы и т.д. Мы проиллюстрируем это, проведя модификацию

написанной ранее программы сортировки так, чтобы при задании

необязательного аргумента -N она бы сортировала строки ввода

численно, а не лексикографически.

Сортировка часто состоит из трех частей - сравнения, ко-

торое определяет упорядочивание любой пары объектов, перес-

тановки, изменяющей их порядок, и алгоритма сортировки, осу-

ществляющего сравнения и перестановки до тех пор, пока

объекты не расположатся в нужном порядке. Алгоритм сортиров-

ки не зависит от операций сравнения и перестановки, так что,

передавая в него различные функции сравнения и перестановки,

мы можем организовать сортировку по различным критериям.

Именно такой подход используется в нашей новой программе

сортировки.

Как и прежде, лексикографическое сравнение двух строк

осуществляется функцией STRCMP, а перестановка функцией

SWAP; нам нужна еще функция NUMCMP, сравнивающая две строки

на основе численного значения и возвращающая условное указа-

ние того же вида, что и STRCMP. Эти три функции описываются

в MAIN и указатели на них передаются в SORT. В свою очередь

функция SORT обращается к этим функциям через их указатели.

мы урезали обработку ошибок в аргументах с тем, чтобы сосре-

доточиться на главных вопросах.

* 125 -

#DEFINE LINES 100 /\* MAX NUMBER OF LINES

TO BE SORTED \*/

MAIN(ARGC, ARGV) /\* SORT INPUT LINES \*/ INT ARGC;

CHAR \*ARGV[];

\(

CHAR **LINEPTR[LINES]; /** POINTERS TO TEXT LINES \*/

INT NLINES; /\* NUMBER OF INPUT LINES READ \*/

INT STRCMP(), NUMCMP(); /\* COMPARSION FUNCTIONS \*/

INT SWAP(); /\* EXCHANGE FUNCTION \*/

INT NUMERIC = 0; /\* 1 IF NUMERIC SORT \*/

IF(ARGC>1 && ARGV[1][0] == '-' && ARGV[1][1]=='N')

NUMERIC = 1;

IF(NLINES = READLINES(LINEPTR, LINES)) >= 0) \(

IF (NUMERIC)

SORT(LINEPTR, NLINES, NUMCMP, SWAP);

ELSE

SORT(LINEPTR, NLINES, STRCMP, SWAP);

WRITELINES(LINEPTR, NLINES);

\) ELSE

PRINTF(“INPUT TOO BIG TO SORT\N”);

\)

Здесь STRCMP, NIMCMP и SWAP - адреса функций; так как извес-

тно, что это функции, операция & здесь не нужна совершенно

аналогично тому, как она не нужна и перед именем массива.

Передача адресов функций организуется компилятором.

Второй шаг состоит в модификации SORT:

SORT(V, N, COMP, EXCH) /\* SORT STRINGS V[0] ... V[N-1] \*/ CHAR **V[]; /** INTO INCREASING ORDER \*/ INT N;

INT (\*COMP)(), (\*EXCH)();

\(

INT GAP, I, J;

FOR(GAP = N/2; GAP > 0; GAP /= 2)

FOR(I = GAP; I < N; I++)

FOR(J = I-GAP; J >= 0; J -= GAP) \(

IF((\*COMP)(V[J], V[J+GAP]) <= 0)

BREAK;

(\*EXCH)(&V[J], &V[J+GAP]);

\)

\)

* 126 -

Здесь следует обратить определенное внимание на описа-

ния. Описание

INT (\*COMP)()

говорит, что COMP является указателем на функцию, которая

возвращает значение типа INT. Первые круглые скобки здесь

необходимы; без них описание

INT \*COMP()

говорило бы, что COMP является функцией, возвращающей указа-

тель на целые, что, конечно, совершенно другая вещь.

Использование COMP в строке

IF (\*COMP)(V[J], V[J+GAP]) <= 0)

полностью согласуется с описанием: COMP - указатель на функ-

цию, \*COMP - сама функция, а

(\*COMP)(V[J], V[J+GAP])

* обращение к ней. Круглые скобки необходимы для правильного

объединения компонентов.

Мы уже приводили функцию STRCMP, сравнивающую две строки

по первому численному значению:

NUMCMP(S1, S2) /\* COMPARE S1 AND S2 NUMERICALLY \*/ CHAR \*S1, \*S2;

\(

DOUBLE ATOF(), V1, V2;

V1 = ATOF(S1);

V2 = ATOF(S2);

IF(V1 < V2)

RETURN(-1);

ELSE IF(V1 > V2)

RETURN(1);

ELSE

RETURN (0);

\)

Заключительный шаг состоит в добавлении функции SWAP,

переставляющей два указателя. Это легко сделать, непосредст-

венно используя то, что мы изложили ранее в этой главе.

* 127 -

SWAP(PX, PY) /\* INTERCHANGE \*PX AND \*PY \*/

CHAR \*PX[], \*PY[];

\(

CHAR \*TEMP;

TEMP = \*PX;

\*PX = \*PY;

\*PY = TEMP;

\)

Имеется множество других необязятельных аргументов, ко-

торые могут быть включены в программу сортировки: некоторые

из них составляют интересные упражнения.

Упражнение 5-11.

Модифицируйте SORT таким образом, чтобы она работала с

меткой -R, указывающей на сортировку в обратном (убывающем)

порядке. Конечно, -R должна работать с -N.

Упражнение 5-12.

Добавьте необязательный аргумент -F, объединяющий вместе

прописные и строчные буквы, так чтобы различие регистров не

учитывалось во время сортировки: данные из верхнего и нижне-

го регистров сортируются вместе, так что буква 'а' прописное

и 'а' строчное оказываются соседними , а не разделенными це-

лым алфавитом.

Упражнение 5-13.

Добавьте необязательный аргумент -D (“словарное упорядо-

чивание”), при наличии которого сравниваются только буквы,

числа и пробелы. Позаботьтесь о том, чтобы эта функция рабо-

тала и вместе с -F.

Упражнение 5-14.

Добавьте возможность обработки полей, так чтобы можно

было сортировать поля внутри строк. Каждое поле должно сор-

тироваться в соответствии с независимым набором необязатель-

ных аргументов. (предметный указатель этой книги сортировал-

ся с помощью аргументов -DF для категории указателя и с -N

для номеров страниц).

* 128 -

6. Структуры.

Структура - это набор из одной или более переменных,

возможно различных типов, сгруппированных под одним именем

для удобства обработки. (В некоторых языках, самый известный

из которых паскаль, структуры называются “записями”).

Традиционным примером структуры является учетная карточ-

ка работающего: “служащий” описывается набором атрибутов та-

ких, как фамилия, имя, отчество (ф.и.о.), адрес, код соци-

ального обеспечения, зарплата и т.д. Некоторые из этих атри-

бутов сами могут оказаться структурами: ф.и.о. Имеет нес-

колько компонент, как и адрес, и даже зарплата.

Структуры оказываются полезными при организации сложных

данных особенно в больших программах, поскольку во многих

ситуациях они позволяют сгруппировать связанные данные таким

образом, что с ними можно обращаться, как с одним целым, а

не как с отдельными объектами. В этой главе мы постараемся

продемонстрировать то, как используются структуры. Програм-

мы, которые мы для этого будем использовать, больше, чем

многие другие в этой книге, но все же достаточно умеренных

размеров.

6.1. Основные сведения.

Давайте снова обратимся к процедурам преобразования даты

из главы 5. Дата состоит из нескольких частей таких, как

день, месяц, и год, и, возможно, день года и имя месяца. Эти

пять переменных можно объеденить в одну структуру вида:

STRUCT DATE \(

INT DAY;

INT MONTH;

INT YEAR;

INT YEARDAY;

CHAR MON\_NAME[4];

\);

Описание структуры, состоящее из заключенного в фигурные

скобки списка описаний, начинается с ключевого слова STRUCT.

За словом STRUCT может следовать необязательное имя, называ-

емое ярлыком структуры (здесь это DATе). Такой ярлык именует

структуры этого вида и может использоваться в дальнейшем как

сокращенная запись подробного описания.

Элементы или переменные, упомянутые в структуре, называ-

ются членами. Ярлыки и члены структур могут иметь такие же

имена, что и обычные переменные (т.е. Не являющиеся членами

структур), поскольку их имена всегда можно различить по кон-

тексту. Конечно, обычно одинаковые имена присваивают только

тесно связанным объектам.

* 129 -

Точно так же, как в случае любого другого базисного ти-

па, за правой фигурной скобкой, закрывающей список членов,

может следовать список переменных.

Оператор

STRUCT \( ...\) X,Y,Z;

синтаксически аналогичен

INT X,Y,Z;

в том смысле, что каждый из операторов описывает X , Y и Z в

качестве переменных соотвествующих типов и приводит к выде-

лению для них памяти.

Описание структуры, за которым не следует списка пере-

менных, не приводит к выделению какой-либо памяти; оно толь-

ко определяет шаблон или форму структуры. Однако, если такое

описание снабжено ярлыком, то этот ярлык может быть исполь-

зован позднее при определении фактических экземпляров струк-

тур. Например, если дано приведенное выше описание DATE, то

STRUCT DATE D;

определяет переменную D в качестве структуры типа DATE.

Внешнюю или статическую структуру можно инициализировать,

поместив вслед за ее определением список инициализаторов для

ее компонент:

STRUCT DATE D=\( 4, 7, 1776, 186, “JUL”\);

Член определенной структуры может быть указан в выраже-

нии с помощью конструкции вида

имя структуры . Член

Операция указания члена структуры “.” связывает имя структу-

ры и имя члена. В качестве примера определим LEAP (признак

високосности года) на основе даты, находящейся в структуре

D,

LEAP = D.YEAR % 4 == 0 && D.YEAR % 100 != 0

\!\! D.YEAR % 400 == 0;

или проверим имя месяца

IF (STRCMP(D.MON\_NAME, “AUG”) == 0) ...

Или преобразуем первый символ имени месяца так, чтобы оно

начиналось со строчной буквы

D.MON\_NAME[0] = LOWER(D.MON\_NAME[0]);

* 130 -

Структуры могут быть вложенными; учетная карточка служа-

щего может фактически выглядеть так:

STRUCT PERSON \(

CHAR NAME[NAMESIZE];

CHAR ADDRESS[ADRSIZE];

LONG ZIPCODE; /\* почтовый индекс \*/

LONG SS\_NUMBER; /\* код соц. Обеспечения \*/

DOUBLE SALARY; /\* зарплата \*/

STRUCT DATE BIRTHDATE; /\* дата рождения \*/

STRUCT DATE HIREDATE; /\* дата поступления

на работу \*/

\);

Структура PERSON содержит две структуры типа DATE . Если мы

определим EMP как

STRUCT PERSON EMP;

то

EMP.BIRTHDATE.MONTH

будет ссылаться на месяц рождения. Операция указания члена

структуры “.” ассоциируется слева направо.

6.2. Структуры и функции.

В языке “C” существует ряд ограничений на использование

структур. Обязательные правила заключаются в том, что единс-

твенные операции, которые вы можете проводить со структура-

ми, состоят в определении ее адреса с помощью операции & и

доступе к одному из ее членов. Это влечет за собой то, что

структуры нельзя присваивать или копировать как целое, и что

они не могут быть переданы функциям или возвращены ими. (В

последующих версиях эти ограничения будут сняты). На указа-

тели структур эти ограничения однако не накладываются, так

что структуры и функции все же могут с удобством работать

совместно. И наконец, автоматические структуры, как и авто-

матические массивы, не могут быть инициализированы; инициа-

лизация возможна только в случае внешних или статических

структур.

Давайте разберем некоторые из этих вопросов, переписав с

этой целью функции перобразования даты из предыдущей главы

так, чтобы они использовали структуры. Так как правила зап-

рещают непосредственную передачу структуры функции, то мы

должны либо передавать отдельно компоненты, либо передать

указатель всей структуры. Первая возможность демонстрируется

на примере функции DAY\_OF\_YEAR, как мы ее написали в главе

5:

D.YEARDAY = DAY\_OF\_YEAR(D.YEAR, D.MONTH, D.DAY);

* 131 -

другой способ состоит в передаче указателя. если мы опишем

HIREDATE как

STRUCT DATE HIREDATE;

и перепишем DAY\_OF\_YEAR нужным образом, мы сможем тогда на-

писать

HIREDATE YEARDAY = DAY\_OF\_YEAR(&HIREDATE);

передавая указатель на HIREDATE функции DAY\_OF\_YEAR . Функ-ция должна быть модифицирована, потому что ее аргумент те-перь является указателем, а не списком переменных.

DAY\_OF\_YEAR(PD) /\* SET DAY OF YEAR FROM MONTH, DAY \*/ STRUCT DATE \*PD;

\(

INT I, DAY, LEAP;

DAY = PD->DAY;

LEAP = PD->YEAR % 4 == 0 && PD->YEAR % 100 != 0

\!\! PD->YEAR % 400 == 0;

FOR (I =1; I < PD->MONTH; I++)

DAY += DAY\_TAB[LEAP][I];

RETURN(DAY);

\)

Описание

STRUCT DATE \*PD;

говорит, что PD является указателем структуры типа DATE.

Запись, показанная на примере

PD->YEAR

является новой. Если P - указатель на структуру, то

P-> член структуры

обращается к конкретному члену. (Операция -> - это знак ми-

нус, за которым следует знак “>”.)

Так как PD указывает на структуру, то к члену YEAR можно

обратиться и следующим образом

(\*PD).YEAR

но указатели структур используются настолько часто, что за-

пись -> оказывается удобным сокращением. Круглые скобки в

(\*PD).YEAR необходимы, потому что операция указания члена

* 132 -

стуктуры старше , чем \* . Обе операции, “->” и “.”, ассоции-

руются слева направо, так что конструкции слева и справа

зквивалентны

P->Q->MEMB (P->Q)->MEMB

EMP.BIRTHDATE.MONTH (EMP.BIRTHDATE).MONTH

Для полноты ниже приводится другая функция, MONTH\_DAY, пере-

писанная с использованием структур.

MONTH\_DAY(PD) /\* SET MONTH AND DAY FROM DAY OF YEAR \*/ STRUCT DATE \*PD;

\(

INT I, LEAP;

LEAP = PD->YEAR % 4 == 0 && PD->YEAR % 100 != 0

\!\! PD->YEAR % 400 == 0;

PD->DAY = PD->YEARDAY;

FOR (I = 1; PD->DAY > DAY\_TAB[LEAP][I]; I++)

PD->DAY -= DAY\_TAB[LEAP][I];

PD->MONTH = I;

\)

Операции работы со структурами “->” и “.” наряду со ()

для списка аргументов и [] для индексов находятся на самом

верху иерархии страшинства операций и, следовательно, связы-

ваются очень крепко. Если, например, имеется описание

STRUCT \(

INT X;

INT \*Y;

\) \*P;

то выражение

++P->X

увеличивает х, а не р, так как оно эквивалентно выражению

++(P->х). Для изменения порядка выполнения операций можно

использовать круглые скобки: (++P)->х увеличивает P до дос-

тупа к х, а (P++)->X увеличивает P после. (круглые скобки в

последнем случае необязательны. Почему ?)

Совершенно аналогично \*P->Y извлекает то, на что указы-

вает Y; \*P->Y++ увеличивает Y после обработки того, на что

он указывает (точно так же, как и \*S++); (\*P->Y)++ увеличи-

вает то, на что указывает Y; \*P++->Y увеличивает P после вы-

борки того, на что указывает Y.

* 133 -

6.3. Массивы сруктур.

Структуры особенно подходят для управления массивами

связанных переменных. Рассмотрим, например, программу подс-

чета числа вхождений каждого ключевого слова языка “C”. Нам

нужен массив символьных строк для хранения имен и массив це-

лых для подсчета. одна из возможностей состоит в использова-

нии двух параллельных массивов KEYWORD и KEYCOUNT:

CHAR \*KEYWORD [NKEYS];

INT KEYCOUNT [NKEYS];

Но сам факт, что массивы параллельны, указывает на возмож-

ность другой организации. Каждое ключевое слово здесь по су-

ществу является парой:

CHAR \*KEYWORD;

INT KEYCOUNT;

и, следовательно, имеется массив пар. Описание структуры

STRUCT KEY \(

CHAR \*KEYWORD;

INT KEYCOUNT;

\) KEYTAB [NKEYS];

оперделяет массив KEYTAB структур такого типа и отводит для

них память. Каждый элемент массива является структурой. Это

можно было бы записать и так:

STRUCT KEY \(

CHAR \*KEYWORD;

INT KEYCOUNT;

\);

STRUCT KEY KEYTAB [NKEYS];

Так как структура KEYTAB фактически содержит постоянный

набор имен, то легче всего инициализировать ее один раз и

для всех членов при определении. Инициализация структур

вполне аналогична предыдущим инициализациям - за определени-

ем следует заключенный в фигурные скобки список инициализа-

торов:

STRUCT KEY \(

CHAR \*KEYWORD;

INT KEYCOUNT;

\) KEYTAB[] =\(

“BREAK”, 0,

“CASE”, 0,

“CHAR”, 0,

“CONTINUE”, 0,

“DEFAULT”, 0,

/\* ... \*/

“UNSIGNED”, 0,

“WHILE”, 0

\);

Инициализаторы перечисляются парами соответственно членам

структуры. Было бы более точно заключать в фигурные скобки

инициализаторы для каждой “строки” или структуры следующим

образом:

\( “BREAK”, 0 \),

\( “CASE”, 0 \),

. . .

* 134 -

Но когда инициализаторы являются простыми переменными или

символьными строками и все они присутствуют, то во внутрен-

них фигурных скобках нет необходимости. Как обычно, компиля-

тор сам вычислит число элементов массива KEYTAB, если иници-

ализаторы присутствуют, а скобки [] оставлены пустыми.

Программа подсчета ключевых слов начинается с определе-

ния массива KEYTAB. ведущая программа читает свой файл вво-

да, последовательно обращаясь к функции GETWORD, которая из-

влекает из ввода по одному слову за обращение. Каждое слово

ищется в массиве KEYTAB с помощью варианта функции бинарного

поиска, написанной нами в главе 3. (Конечно, чтобы эта функ-

ция работала, список ключевых слов должен быть расположен в

порядке возрастания).

#DEFINE MAXWORD 20

MAIN() /\* COUNT “C” KEYWORDS \*/

\(

INT N, T;

CHAR WORD[MAXWORD];

WHILE ((T = GETWORD(WORD,MAXWORD)) != EOF)

IF (T == LETTER)

IF((N = BINARY(WORD,KEYTAB,NKEYS)) >= 0)

KEYTAB[N].KEYCOUNT++;

FOR (N =0; N < NKEYS; N++)

IF (KEYTAB[N].KEYCOUNT > 0)

PRINTF(“%4D %S\N”,

KEYTAB[N].KEYCOUNT, KEYTAB[N].KEYWORD);

\)

BINARY(WORD, TAB, N) /\* FIND WORD IN TAB[0]...TAB[N-1] \*/

CHAR \*WORD;

STRUCT KEY TAB[];

INT N;

\(

INT LOW, HIGH, MID, COND;

LOW = 0;

HIGH = N - 1;

WHILE (LOW <= HIGH) \(

MID = (LOW+HIGH) / 2;

IF((COND = STRCMP(WORD, TAB[MID].KEYWORD)) < 0)

HIGH = MID - 1;

ELSE IF (COND > 0)

LOW = MID + 1;

ELSE

RETURN (MID);

\)

RETURN(-1);

\)

Мы вскоре приведем функцию GETWORD; пока достаточно сказать,

что она возвращает LETTER каждый раз, как она находит слово,

и копирует это слово в свой первый аргумент.

* 135 -

Величина NKEYS - это количество ключевых слов в массиве

KEYTAB . Хотя мы можем сосчитать это число вручную, гораздо

легче и надежнее поручить это машине, особенно в том случае,

если список ключевых слов подвержен изменениям. Одной из

возможностей было бы закончить список инициализаторов указа-

нием на нуль и затем пройти в цикле сквозь массив KEYTAB,

пока не найдется конец.

Но, поскольку размер этого массива полностью определен к

моменту компиляции, здесь имеется более простая возможность.

Число элементов просто есть

SIZE OF KEYTAB / SIZE OF STRUCT KEY

дело в том, что в языке “C” предусмотрена унарная операция

SIZEOF, выполняемая во время компиляции, которая позволяет

вычислить размер любого объекта. Выражение

SIZEOF(OBJECT)

выдает целое, равное размеру указанного объекта. (Размер оп-

ределяется в неспецифицированных единицах, называемых “бай-

тами”, которые имеют тот же размер, что и переменные типа

CHAR). Объект может быть фактической переменной, массивом и

структурой, или именем основного типа, как INT или DOUBLE,

или именем производного типа, как структура. В нашем случае

число ключевых слов равно размеру массива, деленному на раз-

мер одного элемента массива. Это вычисление используется в

утверждении #DEFINE для установления значения NKEYS:

#DEFINE NKEYS (SIZEOF(KEYTAB) / SIZEOF(STRUCT KEY))

Теперь перейдем к функции GETWORD. Мы фактически написа-

ли более общий вариант функции GETWORD, чем необходимо для

этой программы, но он не на много более сложен. Функция

GETWORD возвращает следующее “слово” из ввода, где словом

считается либо строка букв и цифр, начинающихся с буквы, ли-

бо отдельный символ. Тип объекта возвращается в качетве зна-

чения функции; это - LETTER, если найдено слово, EOF для

конца файла и сам символ, если он не буквенный.

GETWORD(W, LIM) /\* GET NEXT WORD FROM INPUT \*/

CHAR \*W;

INT LIM;

\(

INT C, T;

IF (TYPE(C=\*W++=GETCH()) !=LETTER) \(

\*W='\0';

RETURN©;

\)

* 136 -

WHILE (--LIM > 0) \(

T = TYPE(C = \*W++ = GETCH());

IF (T ! = LETTER && T ! = DIGIT) \(

UNGETCH©;

BREAK;

\)

\)

\*(W-1) - '\0';

RETURN(LETTER);

\)

Функция GETWORD использует функции GETCH и UNGETCH, которые

мы написали в главе 4: когда набор алфавитных символов пре-

рывается, функция GETWORD получает один лишний символ. В ре-

зультате вызова UNGETCH этот символ помещается назад во ввод

для следующего обращения.

Функция GETWORD обращается к функции TYPE для определе-

ния типа каждого отдельного символа из файла ввода. Вот ва-

риант, справедливый только для алфавита ASCII.

TYPE© /\* RETURN TYPE OF ASCII CHARACTER \*/ INT C;

\(

IF (C>= 'A' && C<= 'Z' \!\! C>= 'A' && C<= 'Z')

RETURN(LETTER);

ELSE IF (C>= '0' && C<= '9')

RETURN(DIGIT);

ELSE

RETURN©;

\)

Символические константы LETTER и DIGIT могут иметь любые

значения, лишь бы они не вступали в конфликт с символами,

отличными от буквенно-цифровых, и с EOF; очевидно возможен

следующий выбор

#DEFINE LETTER 'A'

#DEFINE DIGIT '0'

функция GETWORD могла бы работать быстрее, если бы обращения

к функции TYPE были заменены обращениями к соответствующему

массиву TYPE[ ]. В стандартной библиотеке языка “C” предус-

мотрены макросы ISALPHA и ISDIGIT, действующие необходимым

образом.

Упражнение 6-1.

Сделайте такую модификацию функции GETWORD и оцените,

как изменится скорость работы программы.

Упражнение 6-2.

Напишите вариант функции TYPE, не зависящий от конкрет-

ного наборасимволов.

* 137 -

Упражнение 6-3.

Напишите вариант программы подсчета ключевых слов, кото-

рый бы не учитывал появления этих слов в заключенных в ка-

вычки строках.

6.4. Указатели на структуры.

Чтобы проиллюстрировать некоторые соображения, связанные

с использованием указателей и массивов структур, давайте

снова составим программу подсчета ключевых строк, используя

на этот раз указатели, а не индексы массивов.

Внешнее описание массива KEYTAB не нужно изменять, но

функции MAIN и BINARY требуют модификации.

MAIN() /\* COUNT C KEYWORD; POINTER VERSION \*/

\(

INT T;

CHAR WORD[MAXWORD];

STRUCT KEY \*BINARY(), \*P;

WHILE ((T = GETWORD(WORD, MAXWORD;) !=EOF)

IF (T==LETTER)

IF ((P=BINARY(WORD,KEYTAB,NKEYS)) !=NULL)

P->KEYCOUNT++;

FOR (P=KEYTAB; P>KEYTAB + NKEYS; P++)

IF (P->KEYCOUNT > 0)

PRINTF(“%4D %S/N”, P->KEYCOUNT, P->KEYWORD);

\)

STRUCT KEY **BINARY(WORD, TAB, N) /** FIND WORD \*/

CHAR **WORD /** IN TAB[0]...TAB[N-1] \*/

STRUCT KEY TAB [];

INT N;

\(

INT COND;

STRUCT KEY \*LOW = &TAB[0];

STRUCT KEY \*HIGH = &TAB[N-1];

STRUCT KEY \*MID;

WHILE (LOW <= HIGH) \(

MID = LOW + (HIGH-LOW) / 2;

IF ((COND = STRCMP(WORD, MID->KEYWORD)) < 0)

HIGH = MID - 1;

ELSE IF (COND > 0)

LOW = MID + 1;

ELSE

RETURN(MID);

\)

RETURN(NULL);

\)

Здесь имеется несколько моментов, которые стоит отме-

тить. Во-первых, описание функции BINARI должно указывать,

что она возвращает указатель на структуру типа KEY, а не на

целое; это объявляется как в функции MAIN, так и в BINARY.

Если функция BINARI находит слово, то она возвращает указа-

тель на него; если же нет, она возвращает NULL.

* 138 -

Во-вторых, все обращения к элементам массива KEYTAB осу-

ществляются через указатели. Это влечет за собой одно сущес-

твенное изменение в функции BINARY: средний элемент больше

нельзя вычислять просто по формуле

MID = (LOW + HIGH) / 2

потому что сложение двух указателей не дает какого-нибудь

полезного результата (даже после деления на 2) и в действи-

тельности является незаконным. эту формулу надо заменить на

MID = LOW + (HIGH-LOW) / 2

в результате которой MID становится указателем на элемент,

расположенный посередине между LOW и HIGH.

Вам также следует разобраться в инициализации LOW и

HIGH. указатель можно инициализировать адресом ранее опреде-

ленного объекта; именно как мы здесь и поступили.

В функции MAIN мы написали

FOR (P=KEYTAB; P < KEYTAB + NKEYS; P++)

Если P является указателем структуры, то любая арифметика с

P учитывает фактический размер данной структуры, так что P++

увеличивает P на нужную величину, в результате чего P указы-

вает на следующий элемент массива структур. Но не считайте,

что размер структуры равен сумме размеров ее членов, - из-за

требований выравнивания для различных объектов в структуре

могут возникать “дыры”.

И, наконец, несколько второстепенный вопрос о форме за-

писи программы. Если возвращаемая функцией величина имеет

тип, как, например, в

STRUCT KEY \*BINARY(WORD, TAB, N)

Tо может оказаться, что имя функции трудно выделить среди

текста. В связи с этим иногда используется другой стиль за-

писи:

STRUCT KEY \*

BINARY(WORD, TAB, N)

Это главным образом дело вкуса; выберите ту форму, которая

вам нравится, и придерживайтесь ее.

6.5. Структуры, ссылающиеся на себя.

Предположим, что нам надо справиться с более общей зада-

чей, состоящей в подсчете числа появлений всех слов в неко-

тором файле ввода. Так как список слов заранее не известен,

мы не можем их упорядочить удобным образом и использовать

бинарный поиск. Мы даже не можем осуществлять последователь-

ный просмотр при поступлении каждого слова, с тем чтобы ус-

тановить, не встречалось ли оно ранее; такая программа будет

работать вечно. (Более точно, ожидаемое время работы растет

как квадрат числа вводимых слов). Как же нам организовать

программу, чтобы справиться со списком произвольных слов?

* 139 -

Одно из решений состоит в том, чтобы все время хранить

массив поступающих до сих пор слов в упорядоченном виде, по-

мещая каждое слово в нужное место по мере их поступления.

OДнако это не следует делать, перемещая слова в линейном

массиве, - это также потребует слишком много времени. Вместо

этого мы используем структуру данных, называемую доичным де-

ревом.

Каждому новому слову соответствует один “узел” дерева;

каждый узел содержит:

указатель текста слова

счетчик числа появлений

указатель узла левого потомка

указатель узла правого потомка

Никакой узел не может иметь более двух детей; возможно от-

сутсвие детей или наличие только одного потомка.

Узлы создаются таким образом, что левое поддерево каждо-

го узла содержит только те слова, которые меньше слова в

этом узле, а правое поддерево только те слова, которые боль-

ше. Чтобы определить, находится ли новое слово уже в дереве,

начинают с корня и сравнивают новое слово со словом, храня-

щимся в этом узле. Если слова совпадают, то вопрос решается

утвердительно. Если новое слово меньше слова в дереве, то

переходят к рассмотрению левого потомка; в противном случае

исследуется правый потомок. Если в нужном направлении пото-

мок отсутствует, то значит новое слово не находится в дереве

и место этого недостающего потомка как раз и является мес-

том, куда следует поместить новое слово. Поскольку поиск из

любого узла приводит к поиску одного из его потомков, то сам

процесс поиска по существу является рекурсивным. В соответс-

твии с этим наиболее естественно использовать рекурсивные

процедуры ввода и вывода.

Возвращаясь назад к описанию узла, ясно, что это будет

структура с четырьмя компонентами:

STRUCT TNODE \( /\* THE BASIC NODE \*/

CHAR **WORD; /** POINTS TO THE TEXT \*/

INT COUNT; /\* NUMBER OF OCCURRENCES \*/

STRUCT TNODE **LEFT; /** LEFT CHILD \*/

STRUCT TNODE **RIGHT; /** RIGHT CHILD \*/

\);

Это “рекурсивное” описание узла может показаться рискован-

ным, но на самом деле оно вполне корректно. Структура не

имеет права содержать ссылку на саму себя, но

STRUCT TNODE \*LEFT;

описывает LEFT как указатель на узел, а не как сам узел.

* 140 -

Текст самой программы оказывается удивительно маленьким,

если, конечно, иметь в распоряжении набор написанных нами

ранее процедур, обеспечивающих нужные действия. Мы имеем в

виду функцию GETWORD для извлечения каждого слова из файла

ввода и функцию ALLOC для выделения места для хранения слов.

Ведущая программа просто считывает слова с помощью функ-

ции GETWORD и помещает их в дерево, используя функцию TREE.

#DEFINE MAXWORD 20

MAIN() /\* WORD FREGUENCY COUNT \*/

\(

STRUCT TNODE \*ROOT, \*TREE();

CHAR WORD[MAXWORD];

INT T;

ROOT = NULL;

WHILE ((T = GETWORD(WORD, MAXWORD)) \! = EOF)

IF (T == LETTER)

ROOT = TREE(ROOT, WORD);

TREEPRINT(ROOT);

\)

Функция TREE сама по себе проста. Слово передается функ-

цией MAIN к верхнему уровню (корню) дерева. На каждом этапе

это слово сравнивается со словом, уже хранящимся в этом уз-

ле, и с помощью рекурсивного обращения к TREE просачивается

вниз либо к левому, либо к правому поддереву. В конце концов

это слово либо совпадает с каким-то словом, уже находящимся

в дереве (в этом случае счетчик увеличивается на единицу),

либо программа натолкнется на нулевой указатель, свидетель-

ствующий о необходимости создания и добавления к дереву но-

вого узла. В случае создания нового узла функция TREE возв-

ращает указатель этого узла, который помещается в родитель-

ский узел.

STRUCT TNODE \*TREE(P, W)

/\* INSTALL W AT OR BELOW P \*/

STRUCT TNODE \*P;

CHAR \*W;

\(

STRUCT TNODE \*TALLOC();

CHAR \*STRSAVE();

INT COND;

IF (P == NULL) \( /\* A NEW WORD

HAS ARRIVED \*/

P == TALLOC(); /\* MAKE A NEW NODE \*/

P->WORD = STRSAVE(W);

P->COUNT = 1;

P->LEFT = P->RIGHT = NULL;

\) ELSE IF ((COND = STRCMP(W, P->WORD)) == 0)

P->COUNT++; /\* REPEATED WORD \*/

ELSE IF (COND < 0)/\* LOWER GOES INTO LEFT SUBTREE \*/

P->LEFT = TREE(P->LEFT, W);

ELSE /\* GREATER INTO RIGHT SUBTREE \*/

P->RIGHT = TREE(P->RIGHT, W);

RETURN(P);

\)

* 141 -

Память для нового узла выделяется функцией TALLOC, явля-

ющейся адаптацией для данного случая функции ALLOC, написан-

ной нами ранее. Она возвращает указатель свободного прост-

ранства, пригодного для хранения нового узла дерева. (Мы

вскоре обсудим это подробнее). Новое слово копируется функ-

цией STRSAVE в скрытое место, счетчик инициализируется еди-

ницей, и указатели обоих потомков полагаются равными нулю.

Эта часть программы выполняется только при добавлении нового

узла к ребру дерева. Мы здесь опустили проверку на ошибки

возвращаемых функций STRSAVE и TALLOC значений (что неразум-

но для практически работающей программы).

Функция TREEPRINT печатает дерево, начиная с левого под-

дерева; в каждом узле сначала печатается левое поддерево

(все слова, которые младше этого слова), затем само слово, а

затем правое поддерево (все слова, которые старше). Если вы

неуверенно оперируете с рекурсией, нарисуйте дерево сами и

напечатайте его с помощью функции TREEPRINT ; это одна из

наиболее ясных рекурсивных процедур, которую можно найти.

TREEPRINT (P) /\* PRINT TREE P RECURSIVELY \*/

STRUCT TNODE \*P;

\(

IF (P != NULL) \(

TREEPRINT (P->LEFT);

PRINTF(“%4D %S\N”, P->COUNT, P->WORD);

TREEPRINT (P->RIGHT);

\)

\)

Практическое замечание: если дерево становится “несба-

лансированным” из-за того, что слова поступают не в случай-

ном порядке, то время работы программы может расти слишком

быстро. В худшем случае, когда поступающие слова уже упоря-

дочены, настоящая программа осуществляет дорогостоящую ими-

тацию линейного поиска. Существуют различные обобщения дво-

ичного дерева, особенно 2-3 деревья и AVL деревья, которые

не ведут себя так “в худших случаях”, но мы не будем здесь

на них останавливаться.

Прежде чем расстаться с этим примером, уместно сделать

небольшое отступление в связи с вопросом о распределении па-

мяти. Ясно, что в программе желательно иметь только один

распределитель памяти, даже если ему приходится размещать

различные виды объектов. Но если мы хотим использовать один

распределитель памяти для обработки запросов на выделение

памяти для указателей на переменные типа CHAR и для указате-

лей на STRUCT TNODE, то при этом возникают два вопроса. Пер-

вый: как выполнить то существующее на большинстве реальных

машин ограничение, что объекты определенных типов должны

удовлетворять требованиям выравнивания (например, часто це-

лые должны размещаться в четных адресах)? Второй: как орга-

низовать описания, чтобы справиться с тем, что функция ALLOC

должна возвращать различные виды указателей ?

* 142 -

Вообще говоря, требования выравнивания легко выполнить

за счет выделения некоторого лишнего пространства, просто

обеспечив то, чтобы распределитель памяти всегда возвращал

указатель, удовлетворяющий всем ограничениям выравнивания.

Например, на PDP-11 достаточно, чтобы функция ALLOC всегда

возвращала четный указатель, поскольку в четный адрес можно

поместить любой тип объекта. единственный расход при этом -

лишний символ при запросе на нечетную длину. Аналогичные

действия предпринимаются на других машинах. Таким образом,

реализация ALLOC может не оказаться переносимой, но ее ис-

пользование будет переносимым. Функция ALLOC из главы 5 не

предусматривает никакого определенного выравнивания; в главе

8 мы продемонстрируем, как правильно выполнить эту задачу.

Вопрос описания типа функции ALLOC является мучительным

для любого языка, который серьезно относится к проверке ти-

пов. Лучший способ в языке “C” - объявить, что ALLOC возвра-

щает указатель на переменную типа CHAR, а затем явно преоб-

разовать этот указатель к желаемому типу с помощью операции

перевода типов. Таким образом, если описать P в виде

CHAR \*P;

то

(STRUCT TNODE \*) P

преобразует его в выражениях в указатель на структуру типа TNODE . Следовательно, функцию TALLOC можно записать в виде:

STRUCT TNODE \*TALLOC()

\(

CHAR \*ALLOC();

RETURN ((STRUCT TNODE \*) ALLOC(SIZEOF(STRUCT TNODE)));

\)

это более чем достаточно для работающих в настоящее время

компиляторов, но это и самый безопасный путь с учетом будую-

щего.

Упражнение 6-4.

Напишите программу, которая читает “C”-программу и печа-

тает в алфавитном порядке каждую группу имен переменных, ко-

торые совпадают в первых семи символах, но отличаются где-то

дальше. (Сделайте так, чтобы 7 было параметром).

Упражнение 6-5.

Напишите программу выдачи перекрестных ссылок, т.е.

Программу, которая печатает список всех слов документа и для

каждого из этих слов печатает список номеров строк, в кото-

рые это слово входит.

Упражнение 6-6.

Напишите программу, которая печатает слова из своего

файла ввода, расположенные в порядке убывания частоты их по-

явления. Перед каждым словом напечатайте число его появле-

ний.

* 143 -

6.6. Поиск в таблице.

Для иллюстрации дальнейших аспектов использования струк-

тур в этом разделе мы напишем программу, представляющую со-

бой содержимое пакета поиска в таблице. Эта программа явля-

ется типичным представителем подпрограмм управления символь-

ными таблицами макропроцессора или компилятора. Рассмотрим,

например, оператор #DEFINE языка “C”. Когда встречается

строка вида

#DEFINE YES 1

то имя YES и заменяющий текст 1 помещаются в таблицу. Позд-

нее, когда имя YES появляется в операторе вида

INWORD = YES;

Oно должно быть замещено на 1.

Имеются две основные процедуры, которые управляют имена-

ми и заменяющими их текстами. Функция INSTALL(S,T) записыва-

ет имя S и заменяющий текст T в таблицу; здесь S и T просто

символьные строки. Функция LOOKUP(S) ищет имя S в таблице и

возвращает либо указатель того места, где это имя найдено,

либо NULL, если этого имени в таблице не оказалось.

При этом используется поиск по алгоритму хеширования -

поступающее имя преобразуется в маленькое положительное чис-

ло, которое затем используется для индексации массива указа-

телей. Элемент массива указывает на начало цепочных блоков,

описывающих имена, которые имеют это значение хеширования.

Если никакие имена при хешировании не получают этого значе-

ния, то элементом массива будет NULL.

Блоком цепи является структура, содержащая указатели на

соответствующее имя, на заменяющий текст и на следующий блок

в цепи. Нулевой указатель следующего блока служит признаком

конца данной цепи.

STRUCT NLIST \( /\* BASIC TABLE ENTRY \*/

CHAR \*NAME;

CHAR \*DEF;

STRUCT NLIST **NEXT; /** NEXT ENTRY IN CHAIN \*/

\);

Массив указателей это просто

DEFINE HASHSIZE 100

TATIC STRUCT NLIST **HASHTAB[HASHSIZE] /** POINTER TABLE \*/

Значение функции хеширования, используемой обеими функ-

циями LOOKUP и INSTALL , получается просто как остаток от

деления суммы символьных значений строки на размер массива.

(Это не самый лучший возможный алгоритм, но его достоинство

состоит в исключительной простоте).

* 144 -

HASH(S) /\* FORM HASH VALUE FOR STRING \*/

CHAR \*S;

\(

INT HASHVAL;

FOR (HASHVAL = 0; \*S != '\0'; )

HASHVAL += \*S++;

RETURN(HASHVAL % HASHSIZE);

\)

В результате процесса хеширования выдается начальный ин-

декс в массиве HASHTAB ; если данная строка может быть

где-то найдена, то именно в цепи блоков, начало которой ука-

зано там. Поиск осуществляется функцией LOOKUP. Если функция

LOOKUP находит, что данный элемент уже присутствует, то она

возвращает указатель на него; если нет, то она возвращает

NULL.

STRUCT NLIST **LOOKUP(S) /** LOOK FOR S IN HASHTAB \*/

CHAR \*S;

\(

STRUCT NLIST \*NP;

FOR (NP = HASHTAB[HASH(S)]; NP != NULL;NP=NP->NEXT)

IF (STRCMP(S, NP->NAME) == 0)

RETURN(NP); /\* FOUND IT \*/

RETURN(NULL); /\* NOT FOUND \*/

Функция INSTALL использует функцию LOOKUP для определе-

ния, не присутствует ли уже вводимое в данный момент имя;

если это так, то новое определение должно вытеснить старое.

В противном случае создается совершенно новый элемент. Если

по какой-либо причине для нового элемента больше нет места,

то функция INSTALL возвращает NULL.

STRUCT NLIST **INSTALL(NAME, DEF) /** PUT (NAME, DEF) \*/

CHAR \*NAME, \*DEF;

\(

STRUCT NLIST \*NP, \*LOOKUP();

CHAR \*STRSAVE(), \*ALLOC();

INT HASHVAL;

IF((NP = LOOKUP(NAME)) == NULL) \( /\* NOT FOUND \*/

NP = (STRUCT NLIST \*) ALLOC(SIZEOF(\*NP));

IF (NP == NULL)

RETURN(NULL);

IF ((NP->NAME = STRSAVE(NAME)) == NULL)

RETURN(NULL);

HASHVAL = HASH(NP->NAME);

NP->NEXT = HASHTAB[HASHVAL];

HASHTAB[HASHVAL] = NP;

\) ELSE /\* ALREADY THERE \*/

FREE((NP->DEF);/\* FREE PREVIOUS DEFINITION \*/

IF ((NP->DEF = STRSAVE(DEF)) == NULL)

RETURN (NULL);

RETURN(NP);

\)

* 145 -

Функция STRSAVE просто копирует строку, указанную в ка-

честве аргумента, в место хранения, полученное в результате

обращения к функции ALLOC. Мы уже привели эту функцию в гла-

ве 5. Так как обращение к функции ALLOC и FREE могут проис-

ходить в любом порядке и в связи с проблемой выравнивания,

простой вариант функции ALLOC из главы 5 нам больше не под-

ходит; смотрите главы 7 и 8.

Упражнение 6-7.

Напишите процедуру, которая будет удалять имя и опреде-

ление из таблицы, управляемой функциями LOOKUP и INSTALL.

Упражнение 6-8.

Разработайте простую, основанную на функциях этого раз-

дела, версию процессора для обработки конструкций #DEFINE ,

пригодную для использования с “C”-программами. Вам могут

также оказаться полезными функции GETCHAR и UNGETCH.

6.7. Поля.

Когда вопрос экономии памяти становится очень существен-

ным, то может оказаться необходимым помещать в одно машинное

слово несколько различных объектов; одно из особенно расп-

росраненных употреблений - набор однобитовых признаков в

применениях, подобных символьным таблицам компилятора. внеш-

не обусловленные форматы данных, такие как интерфейсы аппа-

ратных средств также зачастую предполагают возможность полу-

чения слова по частям.

Представьте себе фрагмент компилятора, который работает

с символьной таблицей. С каждым идентификатором программы

связана определенная информация, например, является он или

нет ключевым словом, является ли он или нет внешним и/или

статическим и т.д. Самый компактный способ закодировать та-

кую информацию - поместить набор однобитовых признаков в от-

дельную переменную типа CHAR или INT.

Обычный способ, которым это делается, состоит в опреде-

лении набора “масок”, отвечающих соответствущим битовым по-

зициям, как в

#DEFINE KEYWORD 01

#DEFINE EXTERNAL 02

#DEFINE STATIC 04

(числа должны быть степенями двойки). Тогда обработка битов

сведется к “жонглированию битами” с помощью операций сдвига,

маскирования и дополнения, описанных нами в главе 2.

Некоторые часто встречающиеся идиомы:

FLAGS \!= EXTERNAL \! STATIC;

включает биты EXTERNAL и STATIC в FLAGS, в то время как

FLAGS &= \^(еXTERNAL \! STATIC);

* 146 -

их выключает, а

IF ((FLAGS & (EXTERNAL \! STATIC)) == 0) ...

истинно, если оба бита выключены.

Хотя этими идиомами легко овладеть, язык “C” в качестве

альтернативы предлагает возможность определения и обработки

полей внутри слова непосредственно, а не посредством побито-

вых логических операций. Поле - это набор смежных битов

внутри одной переменной типа INT. Синтаксис определения и

обработки полей основывается на структурах. Например, сим-

вольную таблицу конструкций #DEFINE, приведенную выше, можно

бы было заменить определением трех полей:

STRUCT \(

UNSIGNED IS\_KEYWORD : 1;

UNSIGNED IS\_EXTERN : 1;

UNSIGNED IS\_STATIC : 1;

\) FLAGS;

Здесь определяется переменная с именем FLAGS, которая содер-

жит три 1-битовых поля. Следующее за двоеточием число задает

ширину поля в битах. Поля описаны как UNSIGNED, чтобы под-

черкнуть, что они действительно будут величинами без знака.

На отдельные поля можно ссылаться, как FLAGS.IS\_STATIE,

FLAGS. IS\_EXTERN, FLAGS.IS\_KEYWORD И т.д., то есть точно так

же, как на другие члены структуры. Поля ведут себя подобно

небольшим целым без знака и могут участвовать в арифметичес-

ких выражениях точно так же, как и другие целые. Таким обра-

зом, предыдущие примеры более естественно переписать так:

FLAGS.IS\_EXTERN = FLAGS.IS\_STATIC = 1;

для включения битов;

FLAGS.IS\_EXTERN = FLAGS.IS\_STATIC = 0;

для выключения битов;

IF (FLAGS.IS\_EXTERN == 0 &&FLAGS.IS\_STATIC == 0)...

для их проверки.

Поле не может перекрывать границу INT; если указанная

ширина такова, что это должно случиться, то поле выравнива-

ется по границе следующего INT. Полям можно не присваивать

имена; неименованные поля (только двоеточие и ширина) ис-

пользуются для заполнения свободного места. Чтобы вынудить

выравнивание на границу следующего INT, можно использовать

специальную ширину 0.

* 147 -

При работе с полями имеется ряд моментов, на которые

следует обратить внимание. По-видимому наиболее существенным

является то, что отражая природу различных аппаратных сред-

ств, распределение полей на некоторых машинах осуществляется

слева направо, а на некоторых справа налево. Это означает,

что хотя поля очень полезны для работы с внутренне опреде-

ленными структурами данных, при разделении внешне определяе-

мых данных следует тщательно рассматривать вопрос о том, ка-

кой конец поступает первым.

Другие ограничения, которые следует иметь в виду: поля

не имеют знака; они могут храниться только в переменных типа

INT (или, что эквивалентно, типа UNSIGNED); они не являются

массивами; они не имеют адресов, так что к ним не применима

операция &.

6.8. Объединения.

Oбъединения - это переменная, которая в различные момен-

ты времени может содержать объекты разных типов и размеров,

причем компилятор берет на себя отслеживание размера и тре-

бований выравнивания. Объединения представляют возможность

работать с различными видами данных в одной области памяти,

не вводя в программу никакой машинно-зависимой информации.

В качестве примера, снова из символьной таблицы компиля-

тора, предположим, что константы могут быть типа INT , FLOAT

или быть указателями на символы. значение каждой конкретной

константы должно храниться в переменной соотвествующего ти-

па, но все же для управления таблицей самым удобным было бы,

если это значение занимало бы один и тот же объем памяти и

хранилось в том же самом месте независимо от его типа. это и

является назначением объединения - выделить отдельную пере-

менную, в которой можно законно хранить любую одну из пере-

менных нескольких типов. Как и в случае полей, синтаксис ос-

новывается на структурах.

UNION U\_TAG \(

INT IVAL;

FLOAT FVAL;

CHAR \*PVAL;

\) UVAL;

Переменная UVAL будет иметь достаточно большой размер,чтобы

хранить наибольший из трех типов, независимо от машины, на

которой осуществляется компиляция, - программа не будет за-

висить от характеристик аппаратных средств. Любой из этих

трех типов может быть присвоен UVAR и затем использован в

выражениях, пока такое использование совместимо: извлекаемый

тип должен совпадать с последним помещенным типом. Дело

программиста - следить за тем, какой тип хранится в объеди-

нении в данный момент; если что-либо хранится как один тип,

а извлекается как другой, то результаты будут зависеть от

используемой машины.

* 149 -

Синтаксически доступ к членам объединения осуществляется

следующим образом:

имя объединения.член

или

указатель объединения ->член

то есть точно так же, как и в случае структур. если для отс-

леживания типа, хранимого в данный момент в UVAL, использу-

ется переменная UTYPE, то можно встретить такой участок

программы:

IF (UTYPE == INT)

PRINTF(“%D\N”, UVAL.IVAL);

ELSE IF (UTYPE == FLOAT)

PRINTF(“%F\N”, UVAL.FVAL);

ELSE IF (UTYPE == STRING)

PRINTF(“%S\N”, UVAL.PVAL);

ELSE

PRINTF(“BAD TYPE %D IN UTYPE\N”, UTYPE);

Объединения могут появляться внутри структур и массивов

и наоборот. Запись для обращения к члену объединения в

структуре (или наоборот) совершенно идентична той, которая

используется во вложенных структурах. например, в массиве

структур, определенным следующим образом

STRUCT \(

CHAR \*NAME;

INT FLAGS;

INT UTYPE;

UNION \(

INT IVAL;

FLOAT FVAL;

CHAR \*PVAL;

\) UVAL;

\) SYMTAB[NSYM];

на переменную IVAL можно сослаться как

SYMTAB[I].UVAL.IVAL

а на первый символ строки PVAL как

\*SYMTAB[I].UVAL.PVAL

В сущности объединение является структурой, в которой все

члены имеют нулевое смещение. Сама структура достаточно ве-

лика, чтобы хранить “самый широкий” член, и выравнивание

пригодно для всех типов, входящих в объединение. Как и в

случае структур, единственными операциями, которые в настоя-

щее время можно проводить с объединениями, являются доступ к

* 150 -

члену и извлечение адреса; объединения не могут быть присво-

ены, переданы функциям или возвращены ими. указатели объеди-

нений можно использовать в точно такой же манере, как и ука-

затели структур.

Программа распределения памяти, приводимая в главе 8 ,

показывает, как можно использовать объединение, чтобы сде-

лать некоторую переменную выровненной по определенному виду

границы памяти.

6.9. Определение типа

В языке “C” предусмотрена возможность, называемая TYPEDEF

для введения новых имен для типов данных. Например, описание

TYPEDEF INT LENGTH;

делает имя LENGTH синонимом для INT. “Тип” LENGTH может быть использован в описаниях, переводов типов и т.д. Точно таким же образом, как и тип INT:

LENGTH LEN, MAXLEN;

LENGTH \*LENGTHS[];

Аналогично описанию

TYPEDEF CHAR \*STRING;

делает STRING синонимом для CHAR\*, то есть для указателя на

символы, что затем можно использовать в описаниях вида

STRING P, LINEPTR[LINES], ALLOC();

Обратите внимание, что объявляемый в конструкции TYPEDEF

тип появляется в позиции имени переменной, а не сразу за

словом TYPEDEF. Синтаксически конструкция TYPEDEF подобна

описаниям класса памяти EXTERN, STATIC и т. Д. мы также ис-

пользовали прописные буквы, чтобы яснее выделить имена.

В качестве более сложного примера мы используем конст-

рукцию TYPEDEF для описания узлов дерева, рассмотренных ра-

нее в этой главе:

TYPEDEF STRUCT TNODE \( /\* THE BASIC NODE \*/

CHAR **WORD; /** POINTS TO THE TEXT \*/

INT COUNT; /\* NUMBER OF OCCURRENCES \*/

STRUCT TNODE **LEFT; /** LEFT CHILD \*/

STRUCT TNODE **RIGHT; /** RIGHT CHILD \*/

\) TREENODE, \*TREEPTR;

В результате получаем два новых ключевых слова: TREENODE

(структура) и TREEPTR (указатель на структуру). Тогда функ-

цию TALLOC можно записать в виде

* 151 -

TREEPTR TALLOC()

\(

CHAR \*ALLOC();

RETURN((TREEPTR) ALLOC(SIZEOF(TREENODE)));

\)

Необходимо подчеркнуть, что описание TYPEDEF не приводит

к созданию нового в каком-либо смысле типа; оно только до-

бавляет новое имя для некоторого существующего типа. при

этом не возникает и никакой новой семантики: описанные таким

способом переменные обладают точно теми же свойствами, что и

переменные, описанные явным образом. По существу конструкция

TYPEDEF сходна с #DEFINE за исключением того, что она интер-

претируется компилятором и потому может осуществлять подста-

новки текста, которые выходят за пределы возможностей мак-

ропроцессора языка “C”. Например,

TYPEDEF INT (\*PFI) ();

создает тип PFI для “указателя функции, возвращающей значе-

ние типа INT”, который затем можно было бы использовать в

программе сортировки из главы 5 в контексте вида

PFI STRCMP, NUMCMP, SWAP;

Имеются две основные причины применения описаний

TYPEDEF. Первая причина связана с параметризацией программы,

чтобы облегчить решение проблемы переносимости. Если для ти-

пов данных, которые могут быть машинно-зависимыми, использо-

вать описание TYPEDEF, то при переносе программы на другую

машину придется изменить только эти описания. Одна из типич-

ных ситуаций состоит в использовании определяемых с помощью

TYPEDEF имен для различных целых величин и в последующем

подходящем выборе типов SHORT, INT и LONG для каждой имею-

щейся машины.

Второе назначение TYPEDEF состоит в обеспечении лучшей доку-

ментации для программы - тип с именем TREEPTR может оказать-

ся более удобным для восприятия, чем тип, который описан

только как указатель сложной структуры.

И наконец, всегда существует вероятность, что в будущем ком-

пилятор или некоторая другая программа, такая как LINT, смо-

жет использовать содержащуюся в описаниях TYPEDEF информацию

для проведения некоторой дополнительной проверки программы.

* 152 -

7. Ввод и вывод

Средства ввода/вывода не являются составной частью языка

“с”, так что мы не выделяли их в нашем предыдущем изложении.

Однако реальные программы взаимодействуют со своей окружаю-

щей средой гораздо более сложным образом, чем мы видели до

сих пор. В этой главе будет описана “стандартная библиотека

ввода/вывода”, то есть набор функций, разработанных для

обеспечения стандартной системы ввода/вывода для “с”- прог-

рамм. Эти функции предназначены для удобства программного

интерфейса, и все же отражают только те операции, которые

могут быть обеспечены на большинстве современных операцион-

ных систем. Процедуры достаточно эффективны для того, чтобы

пользователи редко чувствовали необходимость обойти их “ради

эффективности”, как бы ни была важна конкретная задача. И,

наконец, эти процедуры задуманы быть “переносимыми” в том

смысле, что они должны существовать в совместимом виде на

любой системе, где имеется язык “с”, и что программы, кото-

рые ограничивают свои взаимодействия с системой возможностя-

ми, предоставляемыми стандартной библиотекой, можно будет

переносить с одной системы на другую по существу без измене-

ний.

Мы здесь не будем пытаться описать всю библиотеку вво-

да/вывода; мы более заинтересованы в том, чтобы продемонст-

рировать сущность написания “с”-программ, которые взаимодей-

ствуют со своей операционной средой.

7.1. Обращение к стандартной библиотеке

Каждый исходный файл, который обращается к функции из

стандартной библиотеки, должен вблизи начала содержать стро-

ку

#INCLUDE <STDIO.H>

в файле STDIO.H определяются некоторые макросы и переменные,

используемые библиотекой ввода/вывода. Использование угловых

скобок вместо обычных двойных кавычек - указание компилятору

искать этот файл в справочнике, содержащем заголовки стан-

дартной информации (на системе UNIX обычно LUSRLINELUDE).

Кроме того, при загрузке программы может оказаться необ-

ходимым указать библиотеку явно; на системе PDP-11 UNIX,

например, команда компиляции программы имела бы вид:

CC исходные файлы и т.д. -LS

где -LS указывает на загрузку из стандартной библиотеки.

7.2. Стандартный ввод и вывод - функции GETCHAR и

PUTCHAR

Самый простой механизм ввода заключается в чтении по од-

ному символу за раз из “стандартного ввода”, обычно с терми-

нала пользователя, с помощью функции GETCHAR. Функция

GETCHAR() при каждом к ней обращении возвращает следующий

* 153 -

вводимый символ. В большинстве сред, которые поддерживают

язык “с”, терминал может быть заменен некоторым файлом с по-

мощью обозначения < : если некоторая программа PROG исполь-

зует функцию GETCHAR то командная строка

PROG<INFILE

приведет к тому, что PROG будет читать из файла INFILE, а не

с терминала. Переключение ввода делается таким образом, что

сама программа PROG не замечает изменения; в частности стро-

ка”<INFILE” не включается в командную строку аргументов в

ARGV. Переключение ввода оказывается незаметным и в том слу-

чае, когда вывод поступает из другой программы посредством

поточного (PIPE) механизма; командная строка

OTHERPROG \! PROG

прогоняет две программы, OTHERPROG и PROG, и организует так,

что стандартным вводом для PROG служит стандартный вывод

OTHERPROG.

Функция GETCHAR возвращает значение EOF, когда она попа-

дает на конец файла, какой бы ввод она при этом не считыва-

ла. Стандартная библиотека полагает символическую константу

EOF равной -1 (посредством #DEFINE в файле STDIO.H), но про-

верки следует писать в терминах EOF, а не -1, чтобы избежать

зависимости от конкретного значения.

Вывод можно осуществлять с помощью функции PUTCHAR©,

помещающей символ 'с' в “стандартный ввод”, который по умол-

чанию является терминалом. Вывод можно направить в некоторый

файл с помощью обозначения > : если PROG использует PUTCHAR,

то командная строка

PROG>OUTFILE

приведет к записи стандартного вывода в файл OUTFILE, а не

на терминал. На системе UNIX можно также использовать поточ-

ный механизм. Строка

PROG \! ANOTHERPROG

помещает стандартный вывод PROG в стандартный ввод ANOTHERPROG. И опять PROG не будет осведомлена об изменении направления.

Вывод, осуществляемый функцией PRINTF, также поступает в

стандартный вывод, и обращения к PUTCHAR и PRINTF могут пе-

ремежаться.

Поразительное количество программ читает только из одно-

го входного потока и пишет только в один выходной поток; для

таких программ ввод и вывод с помощью функций GETCHAR,

PUTCHAR и PRINTF может оказаться вполне адекватным и для на-

чала определенно достаточным. Это особенно справедливо тог-

* 154 -

да, когда имеется возможность указания файлов для ввода и

вывода и поточный механизм для связи вывода одной программы

с вводом другой. Рассмотрим, например, программу LOWER, ко-

торая преобразует прописные буквы из своего ввода в строч-

ные:

#INCLUDE <STDIO.H>

MAIN() /\* CONVERT INPUT TO LOWER CASE \*/

\(

INT C;

WHILE ((C = GETCHAR()) != EOF)

PUTCHAR(ISUPPER© ? TOLOWER© : C);

\)

“Функции” ISUPPER и TOLOWER на самом деле являются макроса-

ми, определенными в STDIO.H . Макрос ISUPPER проверяет, яв-

ляется ли его аргумент буквой из верхнего регистра, и возв-

ращает ненулевое значение, если это так, и нуль в противном

случае. Макрос TOLOWER преобразует букву из верхнего регист-

ра в ту же букву нижнего регистра. Независимо от того, как

эти функции реализованы на конкретной машине, их внешнее по-

ведение совершенно одинаково, так что использующие их прог-

раммы избавлены от знания символьного набора.

Если требуется преобразовать несколько файлов, то можно

собрать эти файлы с помощью программы, подобной утилите CAT

системы UNIX,

CAT FILE1 FILE2 ... \! LOWER>OUTPUT

и избежать тем самым вопроса о том, как обратиться к этим

файлам из программы. (Программа CAT приводится позже в этой

главе).

Кроме того отметим, что в стандартной библиотеке вво-

да/вывода “функции” GETCHAR и PUTCHAR на самом деле могут

быть макросами. Это позволяет избежать накладных расходов на

обращение к функции для обработки каждого символа. В главе 8

мы продемонстрируем, как это делается.

7.3. Форматный вывод - функция PRINTF

Две функции: PRINTF для вывода и SCANF для ввода (следу-

ющий раздел) позволяют преобразовывать численные величины в

символьное представлEние и обратно. Они также позволяют ге-

нерировать и интерпретировать форматные строки. Мы уже всюду

в предыдущих главах неформально использовали функцию PRINTF;

здесь приводится более полное и точное описание. Функция

PRINTF(CONTROL, ARG1, ARG2, ...)

* 155 -

преобразует, определяет формат и печатает свои аргументы в

стандартный вывод под управлением строки CONTROL. Управляю-

щая строка содержит два типа объектов: обычные символы, ко-

торые просто копируются в выходной поток, и спецификации

преобразований, каждая из которых вызывает преобразование и

печать очередного аргумента PRINTF.

Каждая спецификация преобразования начинается с символа

% и заканчивается символом преобразования. Между % и симво-

лом преобразования могут находиться:

* знак минус, который указывает о выравнивании преобразован-

ного аргумента по левому краю его поля.

* Строка цифр, задающая минимальную ширину поля. Преобразо-

ванное число будет напечатано в поле по крайней мере этой

ширины, а если необходимо, то и в более широком. Если пре-

образованный аргумент имеет меньше символов, чем указанная

ширина поля, то он будет дополнен слева (или справа, если

было указано выравнивание по левому краю)заполняющими сим-

волами до этой ширины. Заполняющим символом обычно являет-

ся пробел, а если ширина поля указывается с лидирующим ну-

лем, то этим символом будет нуль (лидирующий нуль в данном

случае не означает восьмеричной ширины поля).

* Точка, которая отделяет ширину поля от следующей строки

цифр.

* Строка цифр (точность), которая указывает максимальное

число символов строки, которые должны быть напечатаны, или

число печатаемых справа от десятичной точки цифр для пере-

менных типа FLOAT или DOUBLE.

* Модификатор длины L, который указывает, что соответствую-

щий элемент данных имеет тип LONG, а не INT.

Ниже приводятся символы преобразования и их смысл:

D - аргумент преобразуется к десятичному виду.

O - Аргумент преобразуется в беззнаковую восьмеричную форму

(без лидирующего нуля).

X - Аргумент преобразуется в беззнаковую шестнадцатеричную

форму (без лидирующих 0X).

U - Аргумент преобразуется в беззнаковую десятичную форму.

C - Аргумент рассматривается как отдельный символ.

S - Аргумент является строкой: символы строки печатаются до

тех пор, пока не будет достигнут нулевой символ или не бу-

дет напечатано количество символов, указанное в специфика-

ции точности.

E - Аргумент, рассматриваемый как переменная типа FLOAT или

DOUBLE, преобразуется в десятичную форму в виде

[-]M.NNNNNNE[+-]XX, где длина строки из N определяется

указанной точностью. Точность по умолчанию равна 6.

F - Аргумент, рассматриваемый как переменная типа FLOAT или

DOUBLE, преобразуется в десятичную форму в виде

[-]MMM.NNNNN, где длина строки из N определяется указанной

точностью. Точность по умолчанию равна 6. отметим, что эта

точность не определяет количество печатаемых в формате F

значащих цифр.

* 156 -

G - Используется или формат %E или %F, какой короче; незна-

чащие нули не печатаются.

Если идущий за % символ не является символом преобразования,

то печатается сам этот символ; следовательно,символ % можно

напечатать, указав %%.

Большинство из форматных преобразований очевидно и было

проиллюстрировано в предыдущих главах. Единственным исключе-

нием является то, как точность взаимодействует со строками.

Следующая таблица демонстрирует влияние задания различных

спецификаций на печать “HELLO, WORLD” (12 символов). Мы по-

местили двоеточия вокруг каждого поля для того, чтобы вы

могли видеть его протяженность.

:%10S: :HELLO, WORLD:

:%10-S: :HELLO, WORLD:

:%20S: : HELLO, WORLD:

:%-20S: :HELLO, WORLD :

:%20.10S: : HELLO, WOR:

:%-20.10S: :HELLO, WOR :

:%.10S: :HELLO, WOR:

Предостережение: PRINTF использует свой первый аргумент

для определения числа последующих аргументов и их типов. Ес-

ли количество аргументов окажется недостаточным или они бу-

дут иметь несоответственные типы, то возникнет путаница и вы

получите бессмысленные результаты.

Упражнение 7-1.

Напишите программу, которая будет печатать разумным об-

разом произвольный ввод. Как минимум она должна печатать

неграфические символы в восьмеричном или шестнадцатеричном

виде (в соответствии с принятыми у вас обычаями) и склады-

вать длинные строки.

7.4. Форматный ввод - функция SCANF

Осуществляющая ввод функция SCANF является аналогом

PRINTF и позволяет проводить в обратном направлении многие

из тех же самых преобразований. Функция

SCANF(CONTROL, ARG1, ARG2, ...)

читает символы из стандартного ввода, интерпретирует их в

соответствии с форматом, указанном в аргументе CONTROL, и

помещает результаты в остальные аргументы. Управляющий аргу-

мент описывается ниже; другие аргументы, каждый из которых

должен быть указателем, определяют, куда следует поместить

соответствующим образом преобразованный ввод.

Управляющая строка обычно содержит спецификации преобра-

зования, которые используются для непосредственной интерпре-

тации входных последовательностей. Управляющая строка может

содержать:

* пробелы, табуляции или символы новой строки (“символы пус-

тых промежутков”), которые игнорируются.

* 157 -
* Обычные символы (не %), которые предполагаются совпадающи-

ми со следующими отличными от символов пустых промежутков

символами входного потока.

* Спецификации преобразования, состоящие из символа %, нео-

бязательного символа подавления присваивания \*, необяза-

тельного числа, задающего максимальную ширину поля и сим-

вола преобразования.

Спецификация преобразования управляет преобразованием

следующего поля ввода. нормально результат помещается в пе-

ременную, которая указывается соответствующим аргументом.

Если, однако , с помощью символа \* указано подавление прис-

ваивания, то это поле ввода просто пропускается и никакого

присваивания не производится. Поле ввода определяется как

строка символов, которые отличны от символов простых проме-

жутков; оно продолжается либо до следующего символа пустого

промежутка, либо пока не будет исчерпана ширина поля, если

она указана. Отсюда следует, что при поиске нужного ей вво-

да, функция SCANF будет пересекать границы строк, поскольку

символ новой строки входит в число пустых промежутков.

Символ преобразования определяет интерпретацию поля вво-

да; согласно требованиям основанной на вызове по значению

семантики языка “с” соответствующий аргумент должен быть

указателем. Допускаются следующие символы преобразования:

D - на вводе ожидается десятичное целое; соответствующий ар-

гумент должен быть указателем на целое.

O - На вводе ожидается восьмеричное целое (с лидирующим ну-

лем или без него); соответствующий аргумент должен быть

указателем на целое.

X - На вводе ожидается шестнадцатеричное целое (с лидирующи-

ми 0X или без них); соответствующий аргумент должен быть

указателем на целое.

H - На вводе ожидается целое типа SHORT; соответсвующий ар-

гумент должен быть указателем на целое типа SHORT.

C - Ожидается отдельный символ; соответствующий аргумент

должен быть указателем на символы; следующий вводимый

символ помещается в указанное место. Обычный пропуск сим-

волов пустых промежутков в этом случае подавляется; для

чтения следующего символа, который не является символом

пустого промежутка, пользуйтесь спецификацией преобразо-

вания %1S.

S - Ожидается символьная строка; соответствующий аргумент

должен быть указателем символов, который указывает на

массив символов, который достаточно велик для принятия

строки и добавляемого в конце символа \0.

F - Ожидается число с плавающей точкой; соответствующий ар-

гумент должен быть указателем на переменную типа FLOAT.

Е - символ преобразования E является синонимом для F. Формат

ввода переменной типа FLOAT включает необязательный знак,

строку цифр, возможно содержащую десятичную точку и нео-

бязательное поле экспоненты, состоящее из буквы E, за ко-

торой следует целое, возможно имеющее знак.

* 158 -

Перед символами преобразования D, O и X может стоять L,

которая означает , что в списке аргументов должен находиться

указатель на переменную типа LONG, а не типа INT. Аналогич-

но, буква L может стоять перед символами преобразования E

или F, говоря о том, что в списке аргументов должен нахо-

диться указатель на переменную типа DOUBLE, а не типа FLOAT.

Например, обращение

INT I;

FLOAT X;

CHAR NAME[50];

SCANF(“&D %F %S”, &I, &X, NAME);

со строкой на вводе

25 54.32E-1 THOMPSON

приводит к присваиванию I значения 25,X - значения 5.432 и

NAME - строки “THOMPSON”, надлежащим образом законченной

символом \ 0. эти три поля ввода можно разделить столькими

пробелами, табуляциями и символами новых строк, сколько вы

пожелаете. Обращение

INT I;

FLOAT X;

CHAR NAME[50];

SCANF(“%2D %F %\*D %2S”, &I, &X, NAME);

с вводом

56789 0123 45A72

присвоит I значение 56, X - 789.0, пропустит 0123 и поместит

в NAME строку “45”. при следующем обращении к любой процеду-

ре ввода рассмотрение начнется с буквы A. В этих двух приме-

рах NAME является указателем и, следовательно, перед ним не

нужно помещать знак &.

В качестве другого примера перепишем теперь элементарный

калькулятор из главы 4, используя для преобразования ввода

функцию SCANF:

#INCLUDE <STDIO.H>

MAIN() /\* RUDIMENTARY DESK CALCULATOR \*/

\(

DOUBLE SUM, V;

SUM =0;

WHILE (SCANF(“%LF”, &V) !=EOF)

PRINTF(“\T%.2F\N”, SUM += V);

\)

выполнение функции SCANF заканчивается либо тогда, когда она

исчерпывает свою управляющую строку, либо когда некоторый

элемент ввода не совпадает с управляющей спецификацией. В

качестве своего значения она возвращает число правильно сов-

падающих и присвоенных элементов ввода. Это число может быть

* 159 -

использовано для определения количества найденных элементов

ввода. при выходе на конец файла возвращается EOF; подчерк-

нем, что это значение отлично от 0, что следующий вводимый

символ не удовлетворяет первой спецификации в управляющей

строке. При следующем обращении к SCANF поиск возобновляется

непосредственно за последним введенным символом.

Заключительное предостережение: аргументы функции SCANF

должны быть указателями. Несомненно наиболее распространен-

ная ошибка состоит в написании

SCANF(“%D”, N);

вместо

SCANF(“%D”, &N);

7.5. Форматное преобразование в памяти

От функции SCANF и PRINTF происходят функции SSCANF и

SPRINTF, которые осуществляют аналогичные преобразования, но

оперируют со строкой, а не с файлом. Обращения к этим функ-

циям имеют вид:

SPRINTF(STRING, CONTROL, ARG1, ARG2, ...)

SSCANF(STRING, CONTROL, ARG1, ARG2, ...)

Как и раньше , функция SPRINTF преобразует свои аргументы

ARG1, ARG2 и т.д. В соответствии с форматом, указанным в

CONTROL, но помещает результаты в STRING, а не в стандартный

вывод. KОнечно, строка STRING должна быть достаточно велика,

чтобы принять результат. Например, если NAME - это символь-

ный массив, а N - целое, то

SPRINTF(NAME, “TEMP%D”, N);

создает в NAME строку вида TEMPNNN, где NNN - значение N.

Функция SSCANF выполняет обратные преобразования - она

просматривает строку STRING в соответствии с форматом в ар-

гументе CONTROL и помещает результирующие значения в аргу-

менты ARG1, ARG2 и т.д.эти аргументы должны быть указателя-

ми. В результате обращения

SSCANF(NAME, “TEMP%D”, &N);

переменная N получает значение строки цифр, следующих за

TEMP в NAME.

Упражнение 7-2.

Перепишите настольный калькулятор из главы 4, используя

для ввода и преобразования чисел SCANF и/или SSCANF.

* 160 -

7.6. Доступ к файлам

Все до сих пор написанные программы читали из стандарт-

ного ввода и писали в стандартный вывод, относительно кото-

рых мы предполагали, что они магическим образом предоставле-

ны программе местной операционной системой.

Следующим шагом в вопросе ввода-вывода является написа-

ние программы, работающей с файлом, который не связан зара-

нее с программой. одной из программ, которая явно демонстри-

рует потребность в таких операциях, является CAT, которая

объединяет набор из нескольких именованных файлов в стандар-

тный вывод. Программа CAT используется для вывода файлов на

терминал и в качестве универсального сборщика ввода для

программ, которые не имеют возможности обращаться к файлам

по имени. Например, команда

CAT X.C.Y.C

печатает содержимое файлов X.C и Y.C в стандартный вывод.

Вопрос состоит в том, как организовать чтение из имено-

ванных файлов, т.е., как связать внешние имена, которыми

мыслит пользователь, с фактически читающими данные операто-

рами.

Эти правила просты. Прежде чем можно считывать из неко-

торого файла или записывать в него, этот файл должен быть

открыт с помощью функции FOPEN из стандартной библиотеки.

функция FOPEN берет внешнее имя (подобное X.C или Y.C), про-

водит некоторые обслуживающие действия и переговоры с опера-

ционной системой (детали которых не должны нас касаться) и

возвращает внутреннее имя, которое должно использоваться при

последующих чтениях из файла или записях в него.

Это внутреннее имя, называемое “указателем файла”, фак-

тически является указателем структуры, которая содержит ин-

формацию о файле, такую как место размещения буфера, текущая

позиция символа в буфере, происходит ли чтение из файла или

запись в него и тому подобное. Пользователи не обязаны знать

эти детали, потому что среди определений для стандартного

ввода-вывода, получаемых из файла STDIO.H, содержится опре-

деление структуры с именем FILE. Единственное необходимое

для указателя файла описание демонстрируется примером:

FILE \*FOPEN(), \*FP;

Здесь говорится, что FP является указателем на FILE и

FOPEN возвращает указатель на FILE. Oбратите внимание, что

FILE является именем типа, подобным INT, а не ярлыку струк-

туры; это реализовано как TYPEDEF. (Подробности того, как

все это работает на системе UNIX, приведены в главе 8).

Фактическое обращение к функции FOPEN в программе имеет

вид:

FP=FOPEN(NAME,MODE);

* 161 -

Первым аргументом функции FOPEN является “имя” файла, кото-

рое задается в виде символьной строки. Второй аргумент MODE

(“режим”) также является символьной строкой, которая указы-

вает, как этот файл будет использоваться. Допустимыми режи-

мами являются: чтение (“R”), запись (“W”) и добавление

(“A”).

Если вы откроете файл, который еще не сущетвует, для за-

писи или добавления, то такой файл будет создан (если это

возможно). Открытие существующего файла на запись приводит к

отбрасыванию его старого содержимого. Попытка чтения несу-

ществующего файла является ощибкой. Ошибки могут быть обус-

ловлены и другими причинами (например, попыткой чтения из

файла, не имея на то разрешения). При наличии какой-либо

ошибки функция возвращает нулевое значение указателя NULL

(которое для удобства также определяется в файле STDIO.H).

Другой необходимой вещью является способ чтения или за-

писи, если файл уже открыт. Здесь имеется несколько возмож-

ностей, из которых GETC и PUTC являются простейшими.функция

GETC возвращает следующий символ из файла; ей необходим ука-

затель файла, чтобы знать, из какого файла читать. Таким об-

разом,

C=GETC(FP)

помещает в “C” следующий символ из файла, указанного посред-

ством FP, и EOF, если достигнут конец файла.

Функция PUTC, являющаяся обращением к функции GETC,

PUTC(C,FP)

помещает символ “C” в файл FP и возвращает “C”. Подобно фун-кциям GETCHAR и PUTCHAR, GETC и PUTC могут быть макросами, а не функциями.

При запуске программы автоматически открываются три фай-

ла, которые снабжены определенными указателями файлов. Этими

файлами являются стандартный ввод, стандартный вывод и стан-

дартный вывод ошибок; соответствующие указатели файлов назы-

ваются STDIN, STDOUT и STDERR. Обычно все эти указатели свя-

заны с терминалом, но STDIN и STDOUT могут быть перенаправ-

лены на файлы или в поток (PIPE), как описывалось в разделе

7.2.

Функции GETCHAR и PUTCHAR могут быть определены в терми-

налах GETC, PUTC, STDIN и STDOUT следующим образом:

#DEFINE GETCHAR() GETC(STDIN) #DEFINE PUTCHAR© PUTC(C,

STDOUT)

При работе с файлами для форматного ввода и вывода можно ис-

пользовать функции FSCANF и FPRINTF. Они идентичны функциям

SCANF и PRINTF, за исключением того, что первым аргументом

является указатель файла, определяющий тот файл, который бу-

дет читаться или куда будет вестись запись; управляющая

строка будет вторым аргументом.

* 162 -

Покончив с предварительными замечаниями, мы теперь в

состоянии написать программу CAT для конкатенации файлов.

Используемая здесь основная схема оказывается удобной во

многих программах: если имеются аргументы в командной стро-

ке, то они обрабатываются последовательно. Если такие аргу-

менты отсутствуют, то обрабатывается стандартный ввод. Это

позволяет использовать программу как самостоятельно, так и

как часть большей задачи.

#INCLUDE <STDIO.H>

MAIN(ARGC, ARGV) /\*CAT: CONCATENATE FILES\*/

INT ARGC;

CHAR \*ARGV[];

\(

FILE \*FP, \*FOPEN();

IF(ARGC==1) /\*NO ARGS; COPY STANDARD INPUT\*/

FILECOPY(STDIN);

ELSE

WHILE (--ARGC > 0)

IF ((FP=FOPEN(\*++ARGV,”R”))==NULL) \(

PRINTF(“CAT:CAN'T OPEN %\N”,\*ARGV);

BREAK;

\) ELSE \(

FILECOPY(FP);

FCLOSE(FP);

\)

\)

FILECOPY(FP) /\*COPY FILE FP TO STANDARD OUTPUT\*/

FILE \*FP;

\(

INT C;

WHILE ((C=GETC(FP)) !=EOF)

PUTC(C, STDOUT);

\)

Указатели файлов STDIN и STDOUT заранее определены в библио-

теке ввода-вывода как стандартный ввод и стандартный вывод;

они могут быть использованы в любом месте, где можно исполь-

зовать объект типа FILE\*.они однако являются константами, а

не переменными, так что не пытайтесь им что-либо присваи-

вать.

Функция FCLOSE является обратной по отношению к FOPEN;

она разрывает связь между указателем файла и внешним именем,

установленную функцией FOPEN, и высвобождает указатель файла

для другого файла.большинство операционных систем имеют не-

которые ограничения на число одновременно открытых файлов,

которыми может распоряжаться программа. Поэтому, то как мы

поступили в CAT, освободив не нужные нам более объекты, яв-

ляется хорошей идеей. Имеется и другая причина для примене-

ния функции FCLOSE к выходному файлу - она вызывает выдачу

информации из буфера, в котором PUTC собирает вывод. (При

нормальном завершении работы программы функция FCLOSE вызы-

вается автоматически для каждого открытого файла).

* 163 -

7.7. Обработка ошибок - STDERR и EXIT

Обработка ошибок в CAT неидеальна. Неудобство заключает-

ся в том, что если один из файлов по некоторой причине ока-

зывается недоступным, диагностическое сообщение об этом пе-

чатается в конце объединенного вывода. Это приемлемо, если

вывод поступает на терминал, но не годится, если вывод пос-

тупает в некоторый файл или через поточный (PIPELINE) меха-

низм в другую программу.

Чтобы лучше обрабатывать такую ситуацию, к программе

точно таким же образом, как STDIN и STDOUT, присоединяется

второй выходной файл, называемый STDERR. Если это вообще

возможно, вывод, записанный в файле STDERR, появляется на

терминале пользователя, даже если стандартный вывод направ-

ляется в другое место.

Давайте переделаем программу CAT таким образом, чтобы

сообщения об ошибках писались в стандартный файл ошибок.

“INCLUDE <STDIO.H>

MAIN(ARGC,ARGV) /\*CAT: CONCATENATE FILES\*/

INT ARGC;

CHAR \*ARGV[];

\(

FILE \*FP, \*FOPEN();

IF(ARGC==1) /\*NO ARGS; COPY STANDARD INPUT\*/

FILECOPY(STDIN);

ELSE

WHILE (--ARGC > 0)

IF((FP=FOPEN(\*++ARGV,”R#))==NULL) \(

PRINTF(STDERR,

“CAT: CAN'T OPEN,%S\N”, ARGV);

EXIT(1);

\) ELSE \(

FILECOPY(FP);

\)

EXIT(0);

\)

Программа сообщает об ошибках двумя способами. Диагностичес-

кое сообщение, выдаваемое функцией FPRINTF, поступает в

STDERR и, таким образом, оказывается на терминале пользова-

теля, а не исчезает в потоке (PIPELINE) или в выходном фай-

ле.

Программа также использует функцию EXIT из стандартной

библиотеки, обращение к которой вызывает завершение выполне-

ния программы. Аргумент функции EXIT доступен любой програм-

ме, обращающейся к данной функции, так что успешное или неу-

дачное завершение данной программы может быть проверено дру-

гой программой, использующей эту в качестве подзадачи. По

соглашению величина 0 в качетсве возвращаемого значения сви-

детельствует о том, что все в порядке, а различные ненулевые

значения являются признаками нормальных ситуаций.

* 164 -

Функция EXIT вызывает функцию FCLOSE для каждого откры-

того выходного файла, с тем чтобы вывести всю помещенную в

буферы выходную информацию, а затем вызывает функцию \_EXIT.

Функция \_EXIT приводит к немедленному завершению без очистки

каких-либо буферов; конечно, при желании к этой функции мож-

но обратиться непосредственно.

7.8. Ввод и вывод строк

Стандартная библиотека содержит функцию FGETS, совершен-

но аналогичную функции GETLINE, которую мы использовали на

всем протяжении книги. В результате обращения

FGETS(LINE, MAXLINE, FP)

следующая строка ввода (включая символ новой строки) считы-

вается из файла FP в символьный массив LINE; самое большое

MAXLINE\_1 символ будет прочитан. Результирующая строка за-

канчивается символом \ 0. Нормально функция FGETS возвращает

LINE; в конце файла она возвращает NULL. (Наша функция

GETLINE возвращает длину строки, а при выходе на конец файла

* нуль).

Предназначенная для вывода функция FPUTS записывает

строку (которая не обязана содержать символ новой строки) в

файл:

FPUTS(LINE, FP)

Чтобы показать, что в функциях типа FGETS и FPUTS нет ничего таинственного, мы приводим их ниже, скопированными непосредственно из стандартной библиотеки ввода-вывода:

#INCLUDE <STDIO.H>

CHAR \*FGETS(S,N,IOP) /\*GET AT MOST N CHARS FROM IOP\*/

CHAR \*S;

INT N;

REGISTER FILE \*IOP;

\(

REGISTER INT C;

REGISTER CHAR \*CS;

CS = S;

WHILE(--N>0&&(C=GETC(IOP)) !=EOF)

IF ((\*CS++ = C)=='\N')

BREAK;

\*CS = '\0';

RETURN((C==EOF && CS==S) 7 NULL : S);

\)

FPUTS(S,IOP) /\*PUT STRING S ON FILS IOP\*/

REGISTER CHAR \*S;

REGISTER FILE \*IOP;

\(

REGISTER INT C;

WHILE (C = \*S++)

PUTC(C,IOP);

\)

* 165 -

Упражнение 7-3.

Напишите программу сравнения двух файлов, которая будет

печатать первую строку и позицию символа, где они различают-

ся.

Упражнение 7-4.

Переделайте программу поиска заданной комбинации симво-

лов из главы 5 таким образом, чтобы в качестве ввода исполь-

зовался набор именованных файлов или, если никакие файлы не

указаны как аргументы, стандартный ввод. Следует ли печатать

имя файла при нахождении подходящей строки?

Упражнение 7-5.

Напишите программу печати набора файлов, которая начина-

ет каждый новый файл с новой страницы и печатает для каждого

файла заголовок и счетчик текущих страниц.

7.9. Несколько разнообразных функций

Стандартная библиотека предоставляет множество разнооб-

разных функций, некоторые из которых оказываются особенно

полезными. Мы уже упоминали функции для работы со строками:

STRLEN, STRCPY, STRCAT и STRCMP. Вот некоторые другие.

7.9.1. Проверка вида символов и преобразования

Некоторые макросы выполняют проверку символов и преобра-

зования:

SALPHA© не 0, если “C” алфавитный символ,

0 - если нет.

SUPPER© Не 0, если “C” буква верхнего регистра,

0 - если нет.

SLOWER© Не 0, если “C” буква нижнего регистра,

0 - если нет.

SDIGIT© Не 0, если “C” цифра,

0 - если нет.

SSPACL© Не 0, если “C” пробел, табуляция

или новая строка, 0 - если нет.

OUPPER© Преобразует “C” в букву верхнего регистра.

OLOWER© Преобразует “C” в букву нижнего регистра.

7.9.2. Функция UNGETC

Стандартная библиотека содержит довольно ограниченную

версию функции UNGETCH, написанной нами в главе 4; она назы-

вается UNGETC. В результате обращения

UNGETC(C,FP)

символ “C” возвращается в файл FP. Позволяется возвращать в

каждый файл только один символ. Функция UNGETC может быть

использована в любой из функций ввода и с макросами типа

SCANF, GETC или GETCHAR.

* 166 -

7.9.3. Обращение к системе

Функция SYSTEM(S) выполняет команду, содержащуюся в сим-

вольной строке S, и затем возобновляет выполнение текущей

программы. Содержимое S сильно зависит от используемой опе-

рационной системы. В качестве тривиального примера, укажем,

что на системе UNIX строка

SYSTEM(“DATE”);

приводит к выполнению программы DATE, которая печатает дату

и время дня.

7.9.4. Управление памятью

Функция CALLOC весьма сходна с функцией ALLOC, использо-

ванной нами в предыдущих главах. В результате обращения

CALLOC(N, SIZEOF(OBJCCT))

возвращается либо указатель пространства, достаточного для

размещения N объектов указанного размера, либо NULL, если

запрос не может быть удволетворен. Отводимая память инициа-

лизируется нулевыми значениями.

Указатель обладает нужным для рассматриваемых объектов

выравниванием, но ему следует приписывать соответствующий

тип, как в

CHAR \*CALLOC();

INT \*IP;

IP=(INT\*) CALLOC(N,SIZEOF(INT));

Функция CFREE(P) освобождает пространство, на которое

указывает “P”, причем указатель “P” певоначально должен быть

получен в результате обращения к CALLOC. Здесь нет никаких

ограничений на порядок освобождения пространства, но будет

неприятнейшей ошибкой освободить что-нибудь, что не было по-

лучено обращением к CALLOC.

Реализация программы распределения памяти, подобной

CALLOC, в которой размещенные блоки могут освобождаться в

произвольном порядке, продемонстрирована в главе 8.

* 167 -

8. Интерфейс системы UNIX

Материал этой главы относится к интерфейсу между с-прог-

раммами и операционной системой UNIX. Так как большинство

пользователей языка “C” работают на системе UNIX, эта глава

окажется полезной для большинства читателей. даже если вы

используете с-компилятор на другой машине, изучение приводи-

мых здесь примеров должно помочь вам глубже проникнуть в ме-

тоды программирования на языке “C”.

Эта глава делится на три основные части: ввод/вывод,

система файлов и распределение памяти. Первые две части

предполагают небольшое знакомство с внешними характеристика-

ми системы UNIX.

В главе 7 мы имели дело с системным интерфейсом, который

одинаков для всего многообразия операционных систем. На каж-

дой конкретной системе функции стандартной библиотеки должны

быть написаны в терминах ввода-вывода, доступных на данной

машине. В следующих нескольких разделах мы опишем основную

систему связанных с вводом и выводом точек входа операцион-

ной системы UNIX и проиллюстрируем, как с их помощью могут

быть реализованы различные части стандартной библиотеки.

8.1. Дескрипторы файлов

В операционной системе UNIX весь ввод и вывод осуществ-

ляется посредством чтения файлов или их записи, потому что

все периферийные устройства, включая даже терминал пользова-

теля, являются файлами определенной файловой системы. Это

означает, что один однородный интерфейс управляет всеми свя-

зями между программой и периферийными устройствами.

В наиболее общем случае перед чтением из файла или за-

писью в файл необходимо сообщить системе о вашем намерении;

этот процесс называется “открытием” файла. Система выясня-

ет,имеете ли вы право поступать таким образом (существует ли

этот файл? имеется ли у вас разрешение на обращение к не-

му?), и если все в порядке, возвращает в программу небольшое

положительное целое число, называемое дескриптором файла.

всякий раз, когда этот файл используется для ввода или выво-

да, для идентификации файла употребляется дескриптор файла,

а не его имя. (Здесь существует примерная аналогия с исполь-

зованием READ (5,...) и WRITE (6,...) в фортране). Вся ин-

формация об открытом файле содержится в системе; программа

пользователя обращается к файлу только через дескриптор фай-

ла.

Для удобства выполнения обычных операций ввода и вывода

с помощью терминала пользователя существуют специальные сог-

лашения. Когда интерпретатор команд (“SHELL”) прогоняет

программу, он открывает три файла, называемые стандартным

вводом, стандартным выводом и стандартным выводом ошибок,

которые имеют соответственно числа 0, 1 и 2 в качестве деск-

рипторов этих файлов. В нормальном состоянии все они связаны

с терминалом, так что если программа читает с дескриптором

файла 0 и пишет с дескрипторами файлов 1 и 2, то она может

осуществлять ввод и вывод с помощью терминала, не заботясь

об открытии соответствующих файлов.

* 168 -

Пользователь программы может перенаправлять ввод и вывод

на файлы, используя операции командного интерпретатора SHELL

“<” и “>” :

PROG <INFILE>OUTFILE

В этом случае интерпретатор команд SHELL изменит присваива-

ние по умолчанию дескрипторов файлов 0 и 1 с терминала на

указанные файлы. Нормально дескриптор файла 2 остается свя-

занным с терминалом, так что сообщения об ошибках могут пос-

тупать туда. Подобные замечания справедливы и тогда, когда

ввод и вывод связан с каналом. Следует отметить, что во всех

случаях прикрепления файлов изменяются интерпретатором

SHELL, а не программой. Сама программа, пока она использует

файл 0 для ввода и файлы 1 и 2 для вывода, не знает ни отку-

да приходит ее ввод, ни куда поступает ее выдача.

8.2. Низкоуровневый ввод/вывод - операторы READ и WRITE.

Самый низкий уровень ввода/вывода в системе UNIX не пре-

дусматривает ни какой-либо буферизации, ни какого-либо дру-

гого сервиса; он по существу является непосредственным вхо-

дом в операционную систему. Весь ввод и вывод осуществляется

двумя функциями: READ и WRITE. Первым аргументом обеих функ-

ций является дескриптор файла. Вторым аргументом является

буфер в вашей программе, откуда или куда должны поступать

данные. Третий аргумент - это число подлежащих пересылке

байтов. Обращения к этим функциям имеют вид:

N\_READ=READ(FD,BUF,N);

N\_WRITTEN=WRITE(FD,BUF,N);

При каждом обращении возвращается счетчик байтов, указываю-

щий фактическое число переданных байтов. При чтении возвра-

щенное число байтов может оказаться меньше, чем запрошенное

число. Возвращенное нулевое число байтов означает конец фай-

ла, а “-1” указывает на наличие какой-либо ошибки. При запи-

си возвращенное значение равно числу фактически записанных

байтов; несовпадение этого числа с числом байтов, которое

предполагалось записать, обычно свидетельствует об ошибке.

Количество байтов, подлежащих чтению или записи, может

быть совершенно произвольным. Двумя самыми распространенными

величинами являются “1”, которая означает передачу одного

символа за обращение (т.е. Без использования буфера), и

“512”, которая соответствует физическому размеру блока на

многих периферийных устройствах. Этот последний размер будет

наиболее эффективным, но даже ввод или вывод по одному сим-

волу за обращение не будет необыкновенно дорогим.

Объединив все эти факты, мы написали простую программу

для копирования ввода на вывод, эквивалентную программе ко-

пировки файлов, написанной в главе 1. На системе UNIX эта

программа будет копировать что угодно куда угодно, потому

что ввод и вывод могут быть перенаправлены на любой файл или

устройство.

* 169 -

#DEFINE BUFSIZE 512 /\*BEST SIZE FOR PDP-11 UNIX\*/

MAIN() /\*COPY INPUT TO OUTPUT\*/

\(

CHAR BUF[BUFSIZE];

INT N;

WHILE((N=READ(0,BUF,BUFSIZE))>0)

WRITE(1,BUF,N);

\)

Если размер файла не будет кратен BUFSIZE, то при некотором

обращении к READ будет возвращено меньшее число байтов, ко-

торые затем записываются с помощью WRITE; при следующем пос-

ле этого обращении к READ будет возвращен нуль.

Поучительно разобраться, как можно использовать функции

READ и WRITE для построения процедур более высокого уровня,

таких как GETCHAR, PUTCHAR и т.д. Вот, например, вариант

функции GETCHAR, осуществляющий ввод без использования буфе-

ра.

#DEFINE CMASK 0377 /\*FOR MAKING CHAR'S > 0\*/

GETCHAR() /\*UNBUFFERED SINGLE CHARACTER INPUT\*/

\(

CHAR C;

RETURN((READ(0,&C,1)>0 7 & CMASK : EOF);

\)

Переменная “C” должна быть описана как CHAR, потому что фун-

кция READ принимает указатель на символы. Возвращаемый сим-

вол должен быть маскирован числом 0377 для гарантии его по-

ложительности; в противном случае знаковый разряд может сде-

лать его значение отрицательным. (Константа 0377 подходит

для эвм PDP-11, но не обязательно для других машин).

Второй вариант функции GETCHAR осуществляет ввод больши-

ми порциями, а выдает символы по одному за обращение.

#DEFINE CMASK 0377 /\*FOR MAKING CHAR'S>0\*/

#DEFINE BUFSIZE 512

GETCHAR() /\*BUFFERED VERSION\*/

\(

STATIC CHAR BUF[BUFSIZE];

STATIC CHAR \*BUFP = BUF;

STATIC INT N = 0;

IF (N==0) \( /\*BUFFER IS EMPTY\*/

N=READ(0,BUF,BUFSIZE);

BUFP = BUF;

\)

RETURN((--N>=0) ? \*BUFP++ & CMASK : EOF);

\)

8.3. Открытие, создание, закрытие и расцепление

(UNLINK).

Кроме случая, когда по умолчанию определены стандартные

файлы ввода, вывода и ошибок, вы должны явно открывать фай-

лы, чтобы затем читать из них или писать в них. Для этой це-

ли существуют две точки входа: OPEN и CREAT.

* 170 -

Функция OPEN весьма сходна с функцией FOPEN, рассмотрен-

ной в главе 7, за исключением того, что вместо возвращения

указателя файла она возвращает дескриптор файла, который яв-

ляется просто целым типа INT.

INT FD;

FD=OPEN(NAME,RWMODE);

Как и в случае FOPEN, аргумент NAME является символьной

строкой, соответствующей внешнему имени файла. Однако аргу-

мент, определяющий режим доступа, отличен: RWMODE равно: 0 -

для чтения, 1 - для записи, 2 - для чтения и записи. Если

происходит какая-то ошибка, функция OPEN возвращает “-1”; в

противном случае она возвращает действительный дескриптор

файла.

Попытка открыть файл, который не существует, является

ошибкой. Точка входа CREAT предоставляет возможность созда-

ния новых файлов или перезаписи старых. В результате обраще-

ния

FD=CREAT(NAME,PMODE);

возвращает дескриптор файла, если оказалось возможным соз-

дать файл с именем NAME, и “-1” в противном случае. Если

файл с таким именем уже существует, CREAT усечет его до ну-

левой длины; создание файла, который уже существует, не яв-

ляется ошибкой.

Если файл является совершенно новым, то CREAT создает

его с определенным режимом защиты, специфицируемым аргумен-

том PMODE. В системе файлов на UNIX с файлом связываются де-

вять битов защиты информации, которые управляют разрешением

на чтение, запись и выполнение для владельца файла, для

группы владельцев и для всех остальных пользователей. Таким

образом, трехзначное восьмеричное число наиболее удобно для

спецификации разрешений. Например, число 0755 свидетельству-

ет о разрешении на чтение, запись и выполнение для владельца

и о разрешении на чтение и выполнение для группы и всех ос-

тальных.

Для иллюстрации ниже приводится программа копирования

одного файла в другой, являющаяся упрощенным вариантом ути-

литы CP системы UNIX. (Основное упрощение заключается в том,

что наш вариант копирует только один файл и что второй аргу-

мент не должен быть справочником).

#DEFINE NULL 0

#DEFINE BUFSIZE 512

#DEFINE PMODE 0644/\*RW FOR OWNER,R FOR GROUP,OTHERS\*/

MAIN(ARGC,ARGV) /\*CP: COPY F1 TO F2\*/

INT ARGC;

CHAR \*ARGV[];

\(

INT F1, F2, N;

CHAR BUF[BUFSIZE];

* 171 -

IF (ARGC ! = 3)

ERROR(“USAGE:CP FROM TO”, NULL);

IF ((F1=OPEN(ARGV[1],0))== -1)

ERROR(“CP:CAN'T OPEN %S”, ARGV[1]);

IF ((F2=CREAT(ARGV[2],PMODE))== -1)

ERROR(“CP: CAN'T CREATE %S”, ARGV[2]);

WHILE ((N=READ(F1,BUF,BUFSIZE))>0)

IF (WRITE(F2,BUF,N) !=N)

ERROR(“CP: WRITE ERROR”, NULL);

EXIT(0);

\)

ERROR(S1,S2) /\*PRINT ERROR MESSAGE AND DIE\*/

CHAR \*S1, S2;

\(

PRINTF(S1,S2);

PRINTF(“\N”);

EXIT(1);

\)

Существует ограничение (обычно 15 - 25) на количество

файлов, которые программа может иметь открытыми одновремен-

но. В соответствии с этим любая программа, собирающаяся ра-

ботать со многими файлами, должна быть подготовлена к пов-

торному использованию дескрипторов файлов. Процедура CLOSE

прерывает связь между дескриптором файла и открытым файлом и

освобождает дескриптор файла для использования с некоторым

другим файлом. Завершение выполнения программы через EXIT

или в результате возврата из ведущей программы приводит к

закрытию всех открытых файлов.

Функция расцепления UNLINK (FILENAME) удаляет из системы

файлов файл с именем FILENAME ( из данного справочного фай-

ла. Файл может быть сцеплен с другим справочником, возможно,

под другим именем - примеч.переводчика).

Упражнение 8-1.

Перепишите программу CAT из главы 7, используя функции

READ, WRITE, OPEN и CLOSE вместо их эквивалентов из стандар-

тной библиотеки. Проведите эксперименты для определения от-

носительной скорости работы этих двух вариантов.

8.4. Произвольный доступ - SEEK и LSEEK.

Нормально при работе с файлами ввод и вывод осуществля-

ется последовательно: при каждом обращении к функциям READ и

WRITE чтение или запись начинаются с позиции, непосредствен-

но следующей за предыдущей обработанной. Но при необходимос-

ти файл может читаться или записываться в любом произвольном

порядке. Обращение к системе с помощью функции LSEEK позво-

ляет передвигаться по файлу, не производя фактического чте-

ния или записи. В результате обращения

LSEEK(FD,OFFSET,ORIGIN);

* 172 -

текущая позиция в файле с дескриптором FD передвигается на

позицию OFFSET (смещение), которая отсчитывается от места,

указываемого аргументом ORIGIN (начало отсчета). Последующее

чтение или запись будут теперь начинаться с этой позиции.

Аргумент OFFSET имеет тип LONG; FD и ORIGIN имеют тип INT.

Аргумент ORIGIN может принимать значения 0,1 или 2, указывая

на то, что величина OFFSET должна отсчитываться соответст-

венно от начала файла, от текущей позиции или от конца фай-

ла. Например, чтобы дополнить файл, следует перед записью

найти его конец:

LSEEK(FD,0L,2);

чтобы вернуться к началу (“перемотать обратно”), можно напи-

сать:

LSEEK(FD,0L,0);

обратите внимание на аргумент 0L; его можно было бы записать

и в виде (LONG) 0.

Функция LSEEK позволяет обращаться с файлами примерно

так же, как с большими массивами, правда ценой более медлен-

ного доступа. следующая простая функция, например, считывает

любое количество байтов, начиная с произвольного места в

файле.

GET(FD,POS,BUF,N) /\*READ N BYTES FROM POSITION POS\*/

INT FD, N;

LONG POS;

CHAR \*BUF;

\(

LSEEK(FD,POS,0); /\*GET TO POS\*/

RETURN(READ(FD,BUF,N));

\)

В более ранних редакциях, чем редакция 7 системы UNIX,

основная точка входа в систему ввода-вывода называется SEEK.

Функция SEEK идентична функции LSEEK, за исключением того,

что аргумент OFFSET имеет тип INT, а не LONG. в соответствии

с этим, поскольку на PDP-11 целые имеют только 16 битов, ар-

гумент OFFSET, указываемый функции SEEK, ограничен величиной

65535; по этой причине аргумент ORIGIN может иметь значения

3, 4, 5, которые заставляют функцию SEEK умножить заданное

значение OFFSET на 512 (количество байтов в одном физическом

блоке) и затем интерпретировать ORIGIN, как если это 0, 1

или 2 соответственно. Следовательно, чтобы достичь произ-

вольного места в большом файле, нужно два обращения к SEEK:

сначала одно, которое выделяет нужный блок, а затем второе,

где ORIGIN имеет значение 1 и которое осуществляет передви-

жение на желаемый байт внутри блока.

Упражнение 8-2.

Очевидно, что SEEK может быть написана в терминалах

LSEEK и наоборот. напишите каждую функцию через другую.

* 173 -

8.5. Пример - реализация функций FOPEN и GETC.

Давайте теперь на примере реализации функций FOPEN и

GETC из стандартной библиотеки подпрограмм продемонстрируем,

как некоторые из описанных элементов объединяются вместе.

Напомним, что в стандартной библиотеке файлы описыватся

посредством указателей файлов, а не дескрипторов. Указатель

файла является указателем на структуру, которая содержит

несколько элементов информации о файле: указатель буфера,

чтобы файл мог читаться большими порциями; счетчик числа

символов, оставшихся в буфере; указатель следующей позиции

символа в буфере; некоторые признаки, указывающие режим чте-

ния или записи и т.д.; дескриптор файла.

Описывающая файл структура данных содержится в файле

STDIO.H, который должен включаться (посредством #INCLUDE) в

любой исходный файл, в котором используются функции из стан-

дартной библиотеки. Он также включается функциями этой биб-

лиотеки. В приводимой ниже выдержке из файла STDIO.H имена,

предназначаемые только для использования функциями библиоте-

ки, начинаются с подчеркивания, с тем чтобы уменьшить веро-

ятность совпадения с именами в программе пользователя.

DEFINE \_BUFSIZE 512

DEFINE \_NFILE 20 /\*FILES THAT CAN BE HANDLED\*/

TYPEDEF STRUCT \_IOBUF \(

CHAR \*\_PTR; /\*NEXT CHARACTER POSITION\*/

INT \_CNT; /\*NUMBER OF CHARACTERS LEFT\*/

CHAR \*\_BASE; /\*LOCATION OF BUFFER\*/

INT \_FLAG; /\*MODE OF FILE ACCESS\*/

INT \_FD; /\*FILE DESCRIPTOR\*/

) FILE;

XTERN FILE \_IOB[\_NFILE];

DEFINE STDIN (&\_IOB[0])

DEFINE STDOUT (&\_IOB[1])

DEFINE STDERR (&\_IOB[2])

DEFINE \_READ 01 /\* FILE OPEN FOR READING \*/

DEFINE \_WRITE 02 /\* FILE OPEN FOR WRITING \*/

DEFINE \_UNBUF 04 /\* FILE IS UNBUFFERED \*/

DEFINE \_BIGBUF 010 /\* BIG BUFFER ALLOCATED \*/

DEFINE \_EOF 020 /\* EOF HAS OCCURRED ON THIS FILE \*/

DEFINE \_ERR 040 /\* ERROR HAS OCCURRED ON THIS FILE \*/

DEFINE NULL 0

DEFINE EOF (-1)

DEFINE GETC(P) (--(P)->\_CNT >= 0 \

? \*(P)->\_PTR++ & 0377 : \_FILEBUF(P))

DEFINE GETCHAR() GETC(STDIN)

DEFINE PUTC(X,P) (--(P)->\_CNT >= 0 \

? \*(P)->\_PTR++ = (X) : \_FLUSHBUF((X),P))

DEFINE PUTCHAR(X) PUTC(X,STDOUT)

* 174 -

В нормальном состоянии макрос GETC просто уменьшает

счетчик, передвигает указатель и возвращает символ. (Если

определение #DEFINE слишком длинное, то оно продолжается с

помощью обратной косой черты). Если однако счетчик становит-

ся отрицательным, то GETC вызывает функцию \_FILEBUF, которая

снова заполняет буфер, реинициализирует содержимое структуры

и возвращает символ. Функция может предоставлять переносимый

интерфейс и в то же время содержать непереносимые конструк-

ции: GETC маскирует символ числом 0377, которое подавляет

знаковое расширение, осуществляемое на PDP-11, и тем самым

гарантирует положительность всех символов.

Хотя мы не собираемся обсуждать какие-либо детали, мы

все же включили сюда определение макроса PUTC, для того что-

бы показать, что она работает в основном точно также, как и

GETC, обращаясь при заполнении буфера к функции \_FLUSHBUF.

Теперь может быть написана функция FOPEN. Большая часть

программы функции FOPEN связана с открыванием файла и распо-

ложением его в нужном месте, а также с установлением битов

признаков таким образом, чтобы они указывали нужное состоя-

ние. Функция FOPEN не выделяет какой-либо буферной памяти;

это делается функцией \_FILEBUF при первом чтении из файла.

#INCLUDE <STDIO.H>

#DEFINE PMODE 0644 /\*R/W FOR OWNER;R FOR OTHERS\*/

FILE \*FOPEN(NAME,MODE) /\*OPEN FILE,RETURN FILE PTR\*/

REGISTER CHAR \*NAME, \*MODE;

\(

REGISTER INT FD;

REGISTER FILE \*FP;

IF(\*MODE !='R'&&\*MODE !='W'&&\*MODE !='A') \(

FPRINTF(STDERR,”ILLEGAL MODE %S OPENING %S\N”,

MODE,NAME);

EXIT(1);

\)

FOR (FP=\_IOB;FP<\_IOB+\_NFILE;FP++)

IF((FP->\_FLAG & (\_READ \! \_WRITE))==0)

BREAK; /\*FOUND FREE SLOT\*/

IF(FP>=\_IOB+\_NFILE) /\*NO FREE SLOTS\*/

RETURN(NULL);

IF(\*MODE=='W') /\*ACCESS FILE\*/

FD=CREAT(NAME,PMODE);

ELSE IF(\*MODE=='A') \(

IF((FD=OPEN(NAME,1))==-1)

FD=CREAT(NAME,PMODE);

LSEEK(FD,OL,2);

\) ELSE

FD=OPEN(NAME,0);

IF(FD==-1) /\*COULDN'T ACCESS NAME\*/

RETURN(NULL);

FP->\_FD=FD;

FP->\_CNT=0;

FP->\_BASE=NULL;

FP->\_FLAG &=(\_READ \! \_WRITE);

FP->\_FLAG \!=(\*MODE=='R') ? \_READ : \_WRITE;

RETURN(FP);

\)

* 175 -

Функция \_FILEBUF несколько более сложная. Основная труд-

ность заключается в том, что \_FILEBUF стремится разрешить

доступ к файлу и в том случае, когда может не оказаться дос-

таточно места в памяти для буферизации ввода или вывода. ес-

ли пространство для нового буфера может быть получено обра-

щением к функции CALLOC, то все отлично; если же нет, то

\_FILEBUF осуществляет небуферизованный ввод/ вывод, исполь-

зуя отдельный символ, помещенный в локальном массиве.

#INCLUDE <STDIO.H>

\_FILLBUF(FP) /\*ALLOCATE AND FILL INPUT BUFFER\*/

REGISTER FILE \*FP;

(

STATIC CHAR SMALLBUF(NFILE);/\*FOR UNBUFFERED 1/0\*/

CHAR \*CALLOC();

IF((FR->\_FLAG&\_READ)==0\!\!(FP->\_FLAG&(EOF\!\_ERR))\!=0

RETURN(EOF);

WHILE(FP->\_BASE==NULL) /\*FIND BUFFER SPACE\*/

IF(FP->\_FLAG & \_UNBUF) /\*UNBUFFERED\*/

FP->\_BASE=&SMALLBUF[FP->\_FD];

ELSE IF((FP->\_BASE=CALLOC(\_BUFSIZE,1))==NULL)

FP->\_FLAG \!=\_UNBUF; /\*CAN'T GET BIG BUF\*/

ELSE

FP->\_FLAG \!=\_BIGBUF; /\*GOT BIG ONE\*/

FP->\_PTR=FP->\_BASE;

FP->\_CNT=READ(FP->\_FD, FP->\_PTR,

FP->\_FLAG & \_UNBUF ? 1 : \_BUFSIZE);

FF(--FP->\_CNT<0) \(

IF(FP->\_CNT== -1)

FP->\_FLAG \! = \_EOF;

ELSE

FP->\_FLAG \! = \_ ERR;

FP->\_CNT = 0;

RETURN(EOF);

\)

RETURN(\*FP->\_PTR++ & 0377); /\*MAKE CHAR POSITIVE\*/

)

При первом обращении к GETC для конкретного файла счетчик

оказывается равным нулю, что приводит к обращению к

\_FILEBUF. Если функция \_FILEBUF найдет, что этот файл не от-

крыт для чтения, она немедленно возвращает EOF. В противном

случае она пытается выделить большой буфер, а если ей это не

удается, то буфер из одного символа. При этом она заносит в

\_FLAG соответствующую информацию о буферизации.

Раз буфер уже создан, функция \_FILEBUF просто вызывает

функцию READ для его заполнения, устанавливает счетчик и

указатели и возвращает символ из начала буфера.

Единственный оставшийся невыясненным вопрос состоит в

том, как все начинается. Массив \_IOB должен быть определен и

инициализирован для STDIN, STDOUT и STDERR:

* 176 -

FILE \_IOB[NFILE] = \(

(NULL,0,\_READ,0), /\*STDIN\*/

(NULL,0,NULL,1), /\*STDOUT\*/

(NULL,0,NULL,\_WRITE \! \_UNBUF,2) /\*STDERR\*/

);

Из инициализации части \_FLAG этого массива структур видно,

что файл STDIN предназначен для чтения, файл STDOUT - для

записи и файл STDERR - для записи без использования буфера.

Упражнение 8-3.

Перепишите функции FOPEN и \_FILEBUF, используя поля

вместо явных побитовых операций.

Упражнение 8-4.

Разработайте и напишите функции \_FLUSHBUF и FCLOSE.

Упражнение 8-5.

Стандартная библиотека содержит функцию

FSEEK(FP, OFFSET, ORIGIN)

которая идентична функции LSEEK, исключая то, что FP являет-

ся указателем файла, а не дескриптором файла. Напишите

FSEEK. Убедитесь, что ваша FSEEK правильно согласуется с бу-

феризацией, сделанной для других функций библиотеки.

8.6. Пример - распечатка справочников

Иногда требуется другой вид взаимодействия с системой

файлов - определение информации о файле, а не того, что в

нем содержится. Примером может служить команда LS (“список

справочника”) системы UNIX. По этой команде распечатываются

имена файлов из справочника и, необязательно, другая инфор-

мация, такая как размеры, разрешения и т.д.

Поскольку, по крайней мере, на системе UNIX справочник

является просто файлом, то в такой команде, как LS нет ниче-

го особенного; она читает файл и выделяет нужные части из

находящейся там информации. Однако формат информации опреде-

ляется системой, так что LS должна знать, в каком виде все

представляется в системе.

Мы это частично проиллюстрируем при написании программы

FSIZE. Программа FSIZE представляет собой специальную форму

LS, которая печатает размеры всех файлов, указанных в списке

ее аргументов. Если один из файлов является справочником, то

для обработки этого справочника программа FSIZE обращается

сама к себе рекурсивно. если же аргументы вообще отсутству-

ют, то обрабатывается текущий справочник.

Для начала дадим краткий обзор структуры системы файлов.

Справочник - это файл, который содержит список имен файлов и

некоторое указание о том, где они размещаются. Фактически

это указание является индексом для другой таблицы, которую

называют “I - узловой таблицей”. Для файла I-узел - это то,

* 177 -

где содержится вся информация о файле, за исключением его

имени. Запись в справочнике состоит только из двух элемен-

тов: номера I-узла и имени файла. Точная спецификация посту-

пает при включении файла SYS/DIR.H, который содержит

#DEFINE DIRSIZ 14 /\*MAX LENGTH OF FILE NAME\*/

STRUCT DIRECT /\*STRUCTURE OF DIRECTORY ENTRY\*/

\(

INO\_T&\_INO; /\*INODE NUMBER\*/

CHAR &\_NAME[DIRSIZ]; /\*FILE NAME\*/

\);

“Тип” INO\_T - это определяемый посредством TYPEDEF тип,

который описывает индекс I-узловой таблицы. На PDP-11 UNIX

этим типом оказывается UNSIGNED, но это не тот сорт информа-

ции, который помещают внутрь программы: на разных системах

этот тип может быть различным. Поэтому и следует использо-

вать TYPEDEF. Полный набор “системных” типов находится в

файле SYS/TUPES.H.

Функция STAT берет имя файла и возвращает всю содержащу-

юся в I-ом узле информацию об этом файле (или -1, если име-

ется ошибка). Таким образом, в результате

STRUCT STAT STBUF;

CHAR \*NAME;

STAT(NAME,&STBUF);

структура STBUF наполняется информацией из I-го узла о файле с именем NAME. Структура, описывающая возвращаемую функцией STAT информацию, находится в файле SYS/STAT.H и выглядит следующим образом:

STRUCT STAT /\*STRUCTURE RETURNED BY STAT\*/

\(

DEV\_T ST\_DEV; /\* DEVICE OF INODE \*/

INO\_T ST\_INO; /\* INODE NUMBER \*/

SHORT ST\_MODE /\* MODE BITS \*/

SHORT ST\_NLINK; / \*NUMBER OF LINKS TO FILE \*/

SHORT ST\_UID; /\* OWNER'S USER ID \*/

SHORT ST\_GID; /\* OWNER'S GROUP ID \*/

DEV\_T ST\_RDEV; /\* FOR SPECIAL FILES \*/

OFF\_T ST\_SIZE; /\* FILE SIZE IN CHARACTERS \*/

TIME\_T ST\_ATIME; /\* TIME LAST ACCESSED \*/

TIME\_T ST\_MTIME; /\* TIME LAST MODIFIED \*/

TIME\_T ST\_CTIME; /\* TIME ORIGINALLY CREATED \*/

\)

Большая часть этой информации объясняется в комментариях.

Элемент ST.MODE содержит набор флагов, описывающих файл; для

удобства определения флагов также находятся в файле

SYS/STAT.H.

* 178 -

#DEFINE S\_IFMT 0160000 /\* TYPE OF FILE \*/

#DEFINE S\_IFDIR 0040000 /\* DIRECTORY \*/

#DEFINE S\_IFCHR 0020000 /\* CHARACTER SPECIAL \*/

#DEFINE S\_IFBLK 0060000 /\* BLOCK SPECIAL \*/

#DEFINE S\_IFREG 0100000 /\* REGULAR \*/

#DEFINE S\_ISUID 04000 /\* SET USER ID ON EXECUTION \*/

#DEFINE S\_ISGID 02000 /\* SET GROUP ID ON EXECUTION \*/

#DEFINE S\_ISVTX 01000 /\*SAVE SWAPPED TEXT AFTER USE\*/

#DEFINE S\_IREAD 0400 /\* READ PERMISSION \*/

#DEFINE S\_IWRITE 0200 /\* WRITE PERMISSION \*/

#DEFINE S\_IEXEC 0100 /\* EXECUTE PERMISSION \*/

Теперь мы в состоянии написать программу FSIZE. Если по-

лученный от функции STAT режим указывает, что файл не явля-

ется справочником, то его размер уже под рукой и может быть

напечатан непосредственно. Если же он оказывается справочни-

ком, то мы должны обрабатывать этот справочник отдельно для

каждого файла; так как справочник может в свою очередь со-

держать подсправочники, этот процесс обработки является ре-

курсивным.

Как обычно, ведущая программа главным образом имеет дело

с командной строкой аргументов; она передает каждый аргумент

функции FSIZE в большой буфер.

#INCLUDE <STDIO.H.>

#INCLUDE <SYS/TYPES.H> /\*TYPEDEFS\*/

#INCLUDE <SYS/DIR.H> /\*DIRECTORY ENTRY STRUCTURE\*/

#INCLUDE <SYS/STAT.H> /\*STRUCTURE RETURNED BY STAT\*/

#DEFINE BUFSIZE 256

MAIN(ARGC,ARGV) /\*FSIZE:PRINT FILE SIZES\*/

CHAR \*ARGV[];

\(

CHAR BUF[BUFSIZE];

IF(ARGC==1) \( /\*DEFAULT:CURRENT DIRECTORY\*/

ATRCPY(BUF,”.”);

FSIZE(BUF);

\) ELSE

WHILE(--ARGC>0) \(

STRCPY(BUF,\*++ARGV);

FSIZE(BUF);

\)

\)

Функция FSIZE печатает размер файла. Если однако файл

оказывается справочником, то FSIZE сначала вызывает функцию

DIRECTORY для обработки всех указанных в нем файлов. Обрати-

те внимание на использование имен флагов S\_IFMT и \_IFDIR из

файла STAT.H.

* 179 -

FSIZE(NAME) /\*PRINT SIZE FOR NAME\*/

CHAR \*NAME;

\(

STRUCT STAT STBUF;

IF(STAT(NAME,&STBUF)== -1) \(

FPRINTF(STDERR,”FSIZE:CAN'T FIND %S\N”,NAME);

RETURN;

\)

IF((STBUF.ST\_MODE & S\_IFMT)==S\_IFDIR)

DIRECTORY(NAME);

PRINTF(“%8LD %S\N”,STBUF.ST\_SIZE,NAME);

\)

Функция DIRECTORY является самой сложной. Однако значи-

тельная ее часть связана с созданием для обрабатываемого в

данный момент файла его полного имени, по которому можно

восстановить путь в дереве.

DIRECTORY(NAME) /\*FSIZE FOR ALL FILES IN NAME\*/

CHAR \*NAME;

(

STRUCT DIRECT DIRBUF;

CHAR \*NBP, \*NEP;

INT I, FD;

NBP=NAME+STRLEN(NAME);

\*NBP++='/'; /\*ADD SLASH TO DIRECTORY NAME\*/

IF(NBP+DIRSIZ+2>=NAME+BUFSIZE) /\*NAME TOO LONG\*/

RETURN;

IF((FD=OPEN(NAME,0))== -1)

RETURN;

WHILE(READ(FD,(CHAR \*)&DIRBUF,SIZEOF(DIRBUF))>0) \(

IF(DIRBUF.D\_INO==0) /\*SLOT NOT IN USE\*/

CONTINUE;

IF(STRCMP (DIRBUF.D\_NAME,”.”)==0

\!\! STRCMP(DIRBUF.D\_NAME,”..”)==0

CONTINUE; /\*SKIP SELF AND PARENT\*/

FOR (I=0,NEP=NBP;I<DIRSIZ;I++)

\*NEP++=DIRBUF.D\_NAME[I];

\*NEP++='\0';

FSIZE(NAME);

\)

CLOSE(FD);

\*--NBP='\0'; /\*RESTORE NAME\*/

)

Если некоторая дыра в справочнике в настоящее время не

используется (потому что файл был удален), то в соответству-

ющее I-узловое число равно нулю, и эта позиция пропускается.

Каждый справочник также содержит запись в самом себе, назы-

ваемую “.”, и о своем родителе, “..”; они, очевидно, также

должны быть пропущены, а то программа будет работать весьма

и весьма долго.

* 180 -

Хотя программа FSIZE довольно специализированна, она все

же демонстрирует пару важных идей. во-первых, многие прог-

раммы не являются “системными программами”; они только ис-

пользуют информацию, форма или содержание которой определя-

ется операционной системой. Во-вторых, для таких программ

существенно, что представление этой информации входит только

в стандартные “заголовочные файлы”, такие как STAT.H и

DIR.H, и что программы включают эти файлы, а не помещают

фактические описания внутрь самих программ.

8.7. Пример - распределитель памяти.

В главе 5 мы написали бесхитростный вариант функции

ALLOC. Вариант, который мы напишем теперь, не содержит огра-

ничений: обращения к функциям ALLOC и FREE могут перемежать-

ся в любом порядке; когда это необходимо, функция ALLOC об-

ращается к операционной системе за дополнительной памятью.

Кроме того, что эти процедуры полезны сами по себе, они так-

же иллюстрируют некоторые соображения, связанные с написани-

ем машинно-зависимых программ относительно машинно-независи-

мым образом, и показывают практическое применение структур,

объединений и конструкций TYPEDEF.

Вместо того, чтобы выделять память из скомпилированного

внутри массива фиксированного размера, функция ALLOC будет

по мере необходимости обращаться за памятью к операционной

системе. Поскольку различные события в программе могут тре-

бовать асинхронного выделения памяти, то память, управляемая

ALLOC, не может быть непрерывной. В силу этого свободная па-

мять хранится в виде цепочки свободных блоков. Каждый блок

включает размер, указатель следующего блока и саму свободную

память. Блоки упорядочиваются в порядке возрастания адресов

памяти, причем последний блок (с наибольшим адресом) указы-

вает на первый, так что цепочка фактически оказывается коль-

цом.

При поступлении запроса список свободных блоков просмат-

ривается до тех пор, пока не будет найден достаточно большой

блок. Если этот блок имеет в точности требуемый размер, то

он отцепляется от списка и передается пользователю. Если же

этот блок слишком велик, то он разделяется, нужное количест-

во передается пользователю, а остаток возвращается в свобод-

ный список. Если достаточно большого блока найти не удается,

то операционной системой выделяется новый блок, который

включается в список свободных блоков; затем поиск возобнов-

ляется.

Освобождение памяти также влечет за собой просмотр сво-

бодного списка в поиске подходящего места для введения осво-

божденного блока. Если этот освободившийся блок с какой-либо

стороны примыкает к блоку из списка свободных блоков, то они

объединяются в один блок большего размера, так что память не

становится слишком раздробленной. Обнаружить смежные блоки

просто, потому что свободный список содержится в порядке

возрастания адресов.

* 181 -

Одна из проблем, о которой мы упоминали в главе 5, зак-

лючается в обеспечении того, чтобы возвращаемая функцией

ALLOC память была выровнена подходящим образом для тех

объектов, которые будут в ней храниться. Хотя машины и раз-

личаются, для каждой машины существует тип, требующий наи-

больших ограничений по размещению памяти, если данные самого

ограничительного типа можно поместить в некоторый определен-

ный адрес, то это же возможно и для всех остальных типов.

Например, на IBM 360/370,HONEYWELL 6000 и многих других ма-

шинах любой объект может храниться в границах, соответствую-

щим переменным типа DOUBLE; на PDP-11 будут достаточны пере-

менные типа INT.

Свободный блок содержит указатель следующего блока в це-

почке, запись о размере блока и само свободное пространство;

управляющая информация в начале называется заголовком. Для

упрощения выравнивания все блоки кратны размеру заголовка, а

сам заголовок выровнен надлежащим образом. Это достигается с

помощью объединения, которое содержит желаемую структуру за-

головка и образец наиболее ограничительного по выравниванию

типа:

TYPEDEF INT ALIGN; /\*FORCES ALIGNMENT ON PDP-11\*/

UNION HEADER \( /\*FREE BLOCK HEADER\*/

STRUCT \(

UNION HEADER \*PTR; /\*NEXT FREE BLOCK\*/

UNSIGNED SIZE; /\*SIZE OF THIS FREE BLOCK\*/

\) S;

ALIGN X; /\*FORCE ALIGNMENT OF BLOCKS\*/

\);

TYPEDEF UNION HEADER HEADER;

Функция ALLOC округляет требуемый размер в символах до

нужного числа единиц размера заголовка; фактический блок,

который будет выделен, содержит на одну единицу больше,

предназначаемую для самого заголовка, и это и есть значение,

которое записывается в поле SIZE заголовка. Указатель, возв-

ращаемый функцией ALLOC, указывает на свободное пространст-

во, а не на сам заголовок.

STATIC HEADER BASE; /\*EMPTY LIST TO GET STARTED\*/

STATIC HEADER \*ALLOCP=NULL; /\*LAST ALLOCATED BLOCK\*/

CHAR \*ALLOC(NBYTES)/\*GENERAL-PURPOSE STORAGE ALLOCATOR\*/

UNSIGNED NBYTES;

\(

HEADER \*MORECORE();

REGISTER HEADER \*P, \*G;

REGISTER INT NUNITS;

NUNITS=1+(NBYTES+SIZEOF(HEADER)-1)/SIZEOF(HEADER);

IF ((G=ALLOCP)==NULL) \( /\*NO FREE LIST YET\*/

BASE.S PTR=ALLOCP=G=&BASE;

BASE.S.SIZE=0;

\)

* 182 -

FOR (P=G>S.PTR; ; G=P, P=P->S.PTR) \(

IF (P->S.SIZE>=NUNITS) \( /\*BIG ENOUGH\*/

IF (P->S.SIZE==NUNITS) /\*EXACTLY\*/

G->S.PTR=P->S.PTR;

ELSE \( /\*ALLOCATE TAIL END\*/

P->S.SIZE-=NUNITS;

P+=P->S.SIZE;

P->S.SIZE=NUNITS;

\)

ALLOCP=G;

RETURN((CHAR \*)(P+1));

\)

IF(P==ALLOCP) /\*WRAPPED AROUND FREE LIST\*/

IF((P=MORECORE(NUNITS))==NULL)

RETURN(NULL); /\*NONE LEFT\*/

\)

\)

Переменная BASE используется для начала работы. Если

ALLOCP имеет значение NULL, как в случае первого обращения к

ALLOC, то создается вырожденный свободный список: он состоит

из свободного блока размера нуль и указателя на самого себя.

В любом случае затем исследуется свободный список. Поиск

свободного блока подходящего размера начинается с того места

(ALLOCP), где был найден последний блок; такая стратегия по-

могает сохранить однородность диска. Если найден слишком

большой блок, то пользователю предлагается его хвостовая

часть; это приводит к тому, что в заголовке исходного блока

нужно изменить только его размер. Во всех случаях возвращае-

мый пользователю указатель указывает на действительно сво-

бодную область, лежащую на единицу дальше заголовка. Обрати-

те внимание на то, что функция ALLOC перед возвращением “P”

преобразует его в указатель на символы.

Функция MORECORE получает память от операционной систе-

мы. Детали того, как это осуществляется, меняются, конечно,

от системы к системе. На системе UNIX точка входа SBRK(N)

возвращает указатель на “N” дополнительных байтов памя-

ти.(указатель удволетворяет всем ограничениям на выравнива-

ние). Так как запрос к системе на выделение памяти является

сравнительно дорогой операцией, мы не хотим делать это при

каждом обращении к функции ALLOC. Поэтому функция MORECORE

округляет затребованное число единиц до большего значения;

этот больший блок будет затем разделен так, как необходимо.

Масштабирующая величина является параметром, который может

быть подобран в соответствии с необходимостью.

* 183 -

#DEFINE NALLOC 128 /\*#UNITS TO ALLOCATE AT ONCE\*/

STATIC HEADER \*MORECORE(NU) /\*ASK SYSTEM FOR MEMORY\*/

UNSIGNED NU;

\(

CHAR \*SBRK();

REGISTER CHAR \*CP;

REGISTER HEADER \*UP;

REGISTER INT RNU;

RNU=NALLOC\*((NU+NALLOC-1)/NALLOC);

CP=SBRK(RNU\*SIZEOF(HEADER));

IF ((INT)CP==-1) /\*NO SPACE AT ALL\*/

RETURN(NULL);

UP=(HEADER \*)CP;

UP->S.SIZE=RNU;

FREE((CHAR \*)(UP+1));

RETURN(ALLOCP);

\)

Если больше не осталось свободного пространства, то фун-

кция SBRK возвращает “-1”, хотя NULL был бы лучшим выбором.

Для надежности сравнения “-1” должна быть преобразована к

типу INT. Снова приходится многократно использовать явные

преобразования (перевод) типов, чтобы обеспечить определен-

ную независимость функций от деталей представления указате-

лей на различных машинах.

И последнее - сама функция FREE. Начиная с ALLOCP, она

просто просматривает свободный список в поиске места для

введения свободного блока. Это место находится либо между

двумя существующими блоками, либо в одном из концов списка.

В любом случае, если освободившийся блок примыкает к одному

из соседних, смежные блоки объединяются. Следить нужно толь-

ко затем, чтобы указатели указывали на то, что нужно, и что-

бы размеры были установлены правильно.

FREE(AP) /\*PUT BLOCKE AP IN FREE LIST\*/

CHAR \*AP;

\(

REGISTER HEADER \*P, \*G;

P=(HEADER\*)AP-1; /\*POINT TO HEADER\*/

FOR (G=ALLOCP; !(P>G && P>G->S.PTR);G=G->S.PTR)

IF (G>=G->S.PTR && (P>G \!\! P<G->S.PTR))

BREAK; /\*AT ONE END OR OTHER\*/

IF (P+P->S.SIZE==G->S.PTR)\(/\*JOIN TO UPPER NBR\*/

P->S.SIZE += G->S.PTR->S.SIZE;

P->S.PTR = G->S.PTR->S.PTR;

\) ELSE

P->S.PTR = G->S.PTR;

IF (G+G->S.SIZE==P) \( /\*JOIN TO LOWER NBR\*/

G->S.SIZE+=P->S.SIZE;

G->S.PTR=P->S.PTR;

\) ELSE

G->S.PTR=P;

ALLOCP = G;

\)

* 184 -

Хотя распределение памяти по своей сути зависит от ис-

пользуемой машины, приведенная выше программа показывает,

как эту зависимость можно регулировать и ограничить весьма

небольшой частью программы. Использование TYPEDEF и UNION

позволяет справиться с выравниванием (при условии, что функ-

ция SBRK обеспечивает подходящий указатель). Переводы типов

организуют выполнение явного преобразования типов и даже

справляются с неудачно разработанным системным интерфейсом.

И хотя рассмотренные здесь подробности связаны с распределе-

нием памяти, общий подход равным образом применим и к другим

ситуациям.

Упражнение 8-6.

Функция из стандартной библиотеки CALLOC(N,SIZE) возвра-

щает указатель на “N” объектов размера SIZE, причем соответ-

ствующая память инициализируется на нуль. напишите программу

для CALLOC, используя функцию ALLOC либо в качестве образца,

либо как функцию, к которой происходит обращение.

Упражнение 8-7.

Функция ALLOC принимает затребованный размер, не прове-

ряя его правдоподобности; функция FREE полагает, что тот

блок, который она должна освободить, содержит правильное

значение в поле размера. Усовершенствуйте эти процедуры,

затратив больше усилий на проверку ошибок.

Упражнение 8-8.

Напишите функцию BFREE(P,N), которая включает произволь-

ный блок “P” из “N” символов в список свободных блоков, уп-

равляемый функциями ALLOC и FREE. С помощью функции BFREE

пользователь может в любое время добавлять в свободный спи-

сок статический или внешний массив.

* 185 -

9. Приложение А: справочное руководство по языку 'C'

9.1. Введение

Это руководство описывает язык 'с' для компьютеров DEC

PDP-11, HONEYWELL 6000, IBM система/370 и INTERDATA 8/32.

там, где есть расхождения, мы сосредотачиваемся на версии

для PDP-11, стремясь в то же время указать детали, которые

зависят от реализации. За малым исключением, эти расхождения

непосредственно обусловлены основными свойствами используе-

мого аппаратного оборудования; различные компиляторы обычно

вполне совместимы.

10. Лексические соглашения

Имеется шесть классов лексем: идентификаторы, ключевые

слова, константы, строки, операции и другие разделители.

Пробелы, табуляции , новые строки и комментарии (совместно,

“пустые промежутки”), как описано ниже, игнорируются, за ис-

ключением тех случаев, когда они служат разделителями лек-

сем. Необходим какой-то пустой промежуток для разделения

идентификаторов, ключевых слов и констант, которые в против-

ном случае сольются.

Если сделан разбор входного потока на лексемы вплоть до

данного символа, то в качестве следующей лексемы берется са-

мая длинная строка символов, которая еще может представлять

собой лексему.

10.1. Комментарии

Комментарий открывается символами /\* и заканчивается

символами /\*. Комментарии не вкладываются друг в друга.

10.2. Идентификаторы (имена)

Идентификатор - это последовательность букв и цифр; пер-

вый символ должен быть буквой. Подчеркивание \_ считается

буквой. Буквы нижнего и верхнего регистров различаются. зна-

чащими являются не более, чем первые восемь символов, хотя

можно использовать и больше. На внешние идентификаторы, ко-

торые используются различными ассемблерами и загрузчиками,

накладыватся более жесткие ограничения:

DEC PDP-11 7 символов, 2 регистра

HONEYWELL 6000 6 символов, 1 регистр

IBM 360/370 7 символов, 1 регистр

INTERDATA 8/32 8 символов, 2 регистра

* 186 -

10.3. Ключевые слова

Следующие идентификаторы зарезервированы для использова-

ния в качестве ключевых слов и не могут использоваться иным

образом:

INT EXTERN ELSE

CHAR REGISTER FOR

FLOAT TYPEDEF DO

DOUBLE STATIC WHILE

STRUCT GOTO SWITCH

UNION RETURN CASE

LONG SIZEOF DEFAULT

SHORT BREAK ENTRY

UNSIGNED CONTINUE

\*AUTO IF

Ключевое слово ENTRY в настоящее время не используется ка-

ким-либо компилятором; оно зарезервировано для использования

в будущем. В некоторых реализациях резервируется также слова

FORTRAN и ASM

10.4. Константы

Имеется несколько видов констант, которые перечислены ниже.

В пункте 10.6 резюмируются характеристики аппаратных сред-

ств, которые влияют на размеры.

10.4.1. Целые константы

Целая константа, состоящая из последовательности цифр,

считается восьмеричной, если она начинается с 0 (цифра

нуль), и десятичной в противном случае. Цифры 8 и 9 имеют

восьмеричные значения 10 и 11 соответственно. Последователь-

ность цифр, которой предшествуют символы 0х (нуль, х-малень-

кое) или 0х (нуль х-большое), рассматривается как шестнадца-

тиричное целое. Шестнадцатиричные цифры включают буквы от а

(маленькое) или а (большое) до F (маленькое) или F (большое)

со значениями от 10 до 15. Десятичная константа, величина

которой превышает наибольшее машинное целое со знаком, счи-

тается длинной; восмеричная или шестнадцатиричная константа,

которое превышает наибольшее машинное целое без знака, также

считается длинной.

10.4.2. Явные длинные константы

Десятичная, восмеричная или шестнадцатиричная константа,

за которой непосредственно следует L (эль-маленькое) или L

(эль-большое), является длинной константой. Как обсуждается

ниже, на некоторых машинах целые и длинные значения могут

рассматриваться как идентичные.

10.4.3. Символьные константы

Символьная константа - это символ, заключенный в одиноч-

ные кавычки, как, например, 'X'. Значением символьной конс-

танты является численное значение этого символа в машинном

представлении набора символов.

* 187 -

Некоторые неграфические символы, одиночная кавычка ' и

обратная косая черта \ могут быть представлены в соответст-

вии со следующей таблицей условных последовательностей:

новая строка NL/LF/ \N

горизонтальная табуляция HT \T

символ возврата на одну позицию BS \B

возврат каретки CR \R

переход на новую страницу FF \F

обратная косая черта \ \\

одиночная кавычка ' \'

комбинация битов DDD \DDD

Условная последовательность \DDD состоит из обратной ко-

сой черты, за которой следуют 1,2 или 3 восмеричных цифры,

которые рассмативаются как задающие значение желаемого сим-

вола. Специальным случаем этой конструкции является последо-

вательность \0 (за нулем не следует цифра), которая опреде-

ляет символ NUL. если следующий за обратной косой чертой

символ не совпадает с одним из указанных, то обратная косая

черта игнорируется.

10.4.4. Плавающие константы

Плавающая константа состоит из целой части, десятичной

точки, дробной части, буквы E (маленькая) или E (большая) и

целой экспоненты с необязательным знаком. Как целая, так и

дробная часть являются последовательностью цифр. Либо целая,

либо дробная часть (но не обе) может отсутствовать; либо де-

сятичная точка, либо е (маленькая) и экспонента (но не то и

другое одновременно) может отсутствовать. Каждая плавающая

константа считается имеющей двойную точность.

10.5. Строки

Строка - это последовательность символов, заключенная в

двойные кавычки, как, наприимер,”...”. Строка имеет тип

“массив массивов” и класс памяти STATIC (см. Пункт 4 ниже).

Строка инициализирована указанными в ней символами. Все

строки, даже идентично записанные, считаются различными.

Компилятор помещает в конец каждой строки нулевой байт \0, с

тем чтобы просматривающая строку программа могла определить

ее конец. Перед стоящим внутри строки символом двойной ка-

вычки “ должен быть поставлен символ обратной косой черты \;

кроме того, могут использоваться те же условия последова-

тельности, что и в символьных константах. И последнее, об-

ратная косая черта \, за которой непосредственно следует

символ новой строки, игнорируется.

* 188 -

10.6. Характеристики аппаратных средств

Следующая ниже таблица суммирует некоторые свойства ап-

паратного оборудования, которые меняются от машины к машине.

Хотя они и влияют на переносимость программ, на практике они

представляют маленькую проблему, чем это может казаться за-

ранее.

Таблица 1

DEC PDP-11 HONEYWELL IBM 370 INTERDATA 8/32

ASCII ASCII EBCDIC ASCII

CHAR 8 BITS 9 BITS 8 BITS 8 BITS

INT 16 36 32 32

SHORT 16 36 16 16

LONG 32 36 32 32

FLOAT 32 36 32 32

DOUBLE 64 72 64 64

RANGE -38/+38 -38/+38 -76/+76 -76/+76

11. Синтаксическая нотация

В используемой в этом руководстве синтаксической нотации

синтаксические категории выделяются курсивом (прим. перев.:

в настоящее время синтексические категории вместо курсивом

выделяются подчеркиванием), а литерные слова и символы -

жирным шрифтом. Альтернативные категории перечисляются на

отдельных строчках. Необязательный символ, терминальный или

нетерминальный, указывается индексом “необ”, так что

\( выражение

--------- необ \)

указывает на необязательное выражение, заключенное в фигур-

ных скобках. Синтаксис суммируется в пункте 18.

12. Что в имене тебе моем?

Язык “C” основывает интерпретацию идентификатора на двух

признаках идентификатора: его классе памяти и его типе.

Класс памяти определяет место и время хранения памяти, свя-

занной с идентификатором; тип определяет смысл величин, на-

ходящихся в памяти, определенной под идентификатором.

Имеются четыре класса памяти: автоматическая, статичес-

кая, внешняя и регистровая. Автоматические переменные явля-

ются локальными для каждого вызова блока и исчезают при вы-

ходе из этого блока. Статические переменные являются локаль-

ными, но сохраняют свои значения для следующего входа в блок

даже после того, как управление передается за пределы блока.

Внешние переменные существуют и сохраняют свои значения в

течение выполнения всей программы и могут использоваться для

связи между функциями, в том числе и между независимо ском-

пилированными функциями. Регистровые переменные хранятся

(ели это возможно) в быстрых регистрах машины; подобно авто-

матическим переменным они являются локальными для каждого

блока и исчезают при выходе из этого блока.

* 189 -

В языке “C” предусмотрено несколько основных типов

объектов:

объекты, написанные как символы (CHAR), достаточно вели-

ки, чтобы хранить любой член из соответствующего данной реа-

лизации внутреннего набора символов, и если действительный

символ из этого набора символов хранится в символьной пере-

менной, то ее значение эквивалентно целому коду этого симво-

ла. В символьных переменных можно хранить и другие величины,

но реализация будет машинно-зависимой.

Можно использовать до трех размеров целых, описываемых

как SHORT INT, INT и LONG INT. Длинные целые занимают не

меньше памяти, чем короткие, но в конкретной реализации мо-

жет оказаться, что либо короткие целые, либо длинные целые,

либо те и другие будут эквивалентны простым целым. “Простые”

целые имеют естественный размер, предусматриваемый архиитек-

турой используемой машины; другие размеры вводятся для удво-

летворения специальных потребностей.

Целые без знака, описываемые как UNSIGNED, подчиняются

законам арифметики по модулю 2\*\*N, где N - число битов в их

представлении. (На PDP-11 длинные величины без знака не пре-

дусмотрены).

Плавающие одинарной точности (FLOAT) и плавающие двойной

точности (DOUBLE) в некоторых реализациях могут быть синони-

мами.

Поскольку объекты упомянутых выше типов могут быть ра-

зумно интерпретированы как числа, эти типы будут называться

арифметическими. типы CHAR и INT всех размеров совместно бу-

дут называться целочисленными. Типы FLOAT и DOUBLE совместно

будут называться плавающими типами.

Кроме основных арифметических типов существует концепту-

ально бесконечный класс производных типов, которые образуют-

ся из основных типов следующим образом:

массивы объектов большинства типов;

функции, которые возвращают объекты заданного типа;

указатели на объекты данного типа;

структуры, содержащие последовательность объектов

различных типов;

объединения, способные содержать один из нескольких

объектов различных типов.

Вообще говоря, эти методы построения объектов могут при-

меняться рекурсивно.

13. Объекты и L-значения

Объект является доступным обработке участком памяти;

L-значение - это выражение, ссылающееся на объект. Очевидным

примером выражения L-значения является идентификатор. Сущес-

твуют операции, результатом которых являются L-значения; ес-

ли, например, E - выражение указанного типа, то \*E является

выражением L-значения, ссылающимся на объект E. Название

“L-значение” происходит от выражения присваивания E1=E2, в

котором левая часть должна быть выражением L-значения. При

последующем обсуждении каждой операции будет указываться,

ожидает ли она операндов L-значения и выдает ли она L-значе-

ние.

* 190 -

14. Преобразования

Ряд операций может в зависимости от своих операндов вы-

зывать преобразование значение операнда из одного типа в

другой. В этом разделе объясняются результаты, которые сле-

дует ожидать от таких преобразований. В п. 14.6 Подводятся

итоги преобразований, требуемые большинством обычных опера-

ций; эти сведения дополняются необходимым образом при обсуж-

дении каждой операции.

14.1. Символы и целые

Символ или короткое целое можно использовать всюду, где

можно использовать целое. Во всех случаях значение преобра-

зуется к целому. Преобразование более короткого целого к бо-

лее длинному всегда сопровождается знаковым расширением; це-

лые являются величинами со знаком. Осуществляется или нет

знаковое расширение для символов, зависит от используемой

машины, но гарантируется, что член стандартного набора сим-

волов неотрицателен. из всех машин, рассматриваемых в этом

руководстве, только PDP-11 осуществляет знаковое расширение.

область значений символьных переменных на PDP-11 меняется от

* 128 до 127; символы из набора ASC11 имеют положительные

значения. Символьная константа, заданная с помощью восьме-

ричной условной последовательности, подвергается знаковому

расширению и может оказаться отрицательной; например, '\377'

имеет значение -1.

Когда более длинное целое преобразуется в более короткое

или в CHAR, оно обрезается слева; лишние биты просто отбра-

сываются.

14.2. Типы FLOAT и DOUBLE

Вся плавающая арифметика в “C” выполняется с двойной

точностью каждый раз, когда объект типа FLOAT появляется в

выражении, он удлиняется до DOUBLE посредством добавления

нулей в его дробную часть. когда объект типа DOUBLE должен

быть преобразован к типу FLOAT, например, при присваивании,

перед усечением DOUBLE округляется до длины FLOAT.

14.3. Плавающие и целочисленные величины

Преобразование плавающих значений к целочисленному типу

имеет тенденцию быть до некоторой степени машинно-зависимым;

в частности направление усечения отрицательных чисел меняет-

ся от машине к машине. Результат не определен, если значение

не помещается в предоставляемое пространство.

Преобразование целочисленных значений в плавающие выпол-

няется без осложнений. Может произойти некоторая потеря точ-

ности, если для результата не содержится достаточного коли-

чества битов.

14.4. Указатели и целые

Целое или длинное целое может быть прибавлено к указате-

лю или вычтено из него; в этом случае первая величина преоб-

разуется так, как указывается в разделе описания операции

сложения.

* 191 -

Два указателя на объекты одинакового типа могут быть

вычтены; в этом случае результат преобразуется к целому, как

указывается в разделе описания операции вычитания.

14.5. Целое без знака

Всякий раз, когда целое без знака объединяется с простым

целым, простое целое преобразуется в целое без знака и ре-

зультат оказывается целым без знака. Значением является наи-

меньшее целое без знака, соответствующее целому со знаком

(по модулю 2\*\*размер слова). В двоичном дополнительном пред-

ставлении это преобразование является чисто умозрительным и

не изменяет фактическую комбинацию битов.

Когда целое без знака преобразуется к типу LONG, значе-

ние результата совпадает со значением целого без знака. Та-

ким образом, это преобразование сводится к добавлению нулей

слева.

14.6. Арифметические преобразования

Подавляющее большинство операций вызывает преобразование

и определяет типы результата аналогичным образом. Приводимая

ниже схема в дальнейшем будет называться “обычными арифмети-

ческими преобразованиями”.

Сначала любые операнды типа CHAR или SHORT преобразуются в

INT, а любые операнды типа FLOAT преобразуются в DOUBLE.

Затем, если какой-либо операнд имеет тип DOUBLE, то другой

преобразуется к типу DOUBLE, и это будет типом результата.

В противном случае, если какой-либо операнд имеет тип LONG,

то другой операнд преобразуется к типу LONG, и это и будет

типом результата.

В противном случае, если какой-либо операнд имеет тип

UNSIGNED, то другой операнд преобразуется к типу UNSIGNED,

и это будет типом результата.

В противном случае оба операнда будут иметь тип INT, и это

будет типом результата.

15. Выражения

Старшинство операций в выражениях совпадает с порядком

следования основных подразделов настоящего раздела, начиная

с самого высокого уровня старшинства. Так, например, выраже-

ниями, указываемыми в качестве операндов операции +

(п.15.4), Являются выражения, определенные в п.п.15.1-15.3.

Внутри каждого подраздела операции имеет одинаковое старшин-

ство. В каждом подразделе для описываемых там операций ука-

зывается их ассоциативность слева или справа. Старшинство и

ассоциативность всех операций в выражениях резюмируются в

грамматической сводке в п.18.

В противном случае порядок вычислений выражений не опре-

делен. В частности, компилятор считает себя в праве вычис-

лять подвыражения в том порядке, который он находит наиболее

эффективным, даже если эти подвыражения приводят к побочным

эффектам. Порядок, в котором происходят побочные эффекты, не

специфицируется. Выражения, включающие коммутативные и ассо-

циативные операции ( \*,+,&,!,^ ), могут быть переупорядочены

произвольным образом даже при наличии круглых скобок; чтобы

вынудить определенный порядок вычислений, в этом случае не-

обходимо использовать явные промежуточные переменные.

* 192 -

При вычислении выражений обработка переполнения и про-

верка при делении являются машинно-зависимыми. Все существу-

ющие реализации языка “C” игнорируют переполнение целых; об-

работка ситуаций при делении на 0 и при всех особых случаях

с плавающими числами меняется от машины к машине и обычно

выполняется с помощью библиотечной функции.

15.1. Первичные выражения

Первичные выражения, включающие ., ->, индексацию и об-

ращения к функциям, группируются слева направо.

Первичное выражение:

идентификатор

константа

строка

(выражение)

первичное-выражение [выражение]

первичное-выражение (список-выражений нео

первичное-L-значение . Идентификатор

первичное-выражение -> идентификатор

список-выражений:

выражение

список-выражений, выражение

Идентификатор является первичным выражением при условии, что

он описан подходящим образом, как это обсуждается ниже. тип

идентификатора определяется его описанием. Если, однако, ти-

пом идентификатора является “массив ...”, то значением выра-

жения, состоящего из этого идентификатора , является указа-

тель на первый объект в этом массиве, а типом выражения бу-

дет “указатель на ...”. Более того, идентификатор массива не

является выражением L-значения. подобным образом идентифика-

тор, который описан как “функция, возвращающая ...”, за иск-

лючением того случая, когда он используется в позиции имени

функции при обращении, преобразуется в “указатель на функ-

цию, которая возвращает ...”.

Константа является первичным выражением. В зависимости

от ее формы типом константы может быть INT, LONG или DOUBLE.

Строка является первичным выражением. Исходным ее типом

является “массив символов”; но следуя тем же самым правилам,

которые приведены выше для идентификаторов, он модифицирует-

ся в “указатель на символы”, и результатом является указа-

тель на первый символ строки. (имеется исключение в некото-

рых инициализаторах; см. П. 16.6.)

Выражение в круглых скобках является первичным выражени-

ем, тип и значение которого идентичны типу и значению этого

выражения без скобок. Наличие круглых скобок не влияет на

то, является ли выражение L-значением или нет.

* 193 -

Первичное выражение, за которым следует выражение в

квадратных скобках, является первичным выражением. Интуитив-

но ясно, что это выражение с индексом. Обычно первичное вы-

ражение имеет тип “указатель на ...”, индексное выражение

имеет тип INT, а типом результата является “...”. Выражение

E1[E2] по определению идентично выражению \* ((E1) + (E2)).

Все, что необходимо для понимания этой записи, содержится в

этом разделе; вопросы, связанные с понятием идентификаторов

и операций \* и + рассматриваются в п.п. 15.1, 15.2 И 15.4

соответственно; выводы суммируются ниже в п. 22.3.

Обращение к функции является первичным выражением, за

которым следует заключенный в круглые скобки возможно пустой

список выражений, разделенных запятыми, которые и представ-

ляют собой фактические аргументы функции. Первичное выраже-

ние должно быть типа “функция, возвращающая ...”, а резуль-

тат обращения к функции имеет тип “...”. Как указывается ни-

же, ранее не встречавщийся идентификатор, за которым непос-

редственно следует левая круглая скобка, считается описанным

по контексту, как представляющий функцию, возвращающую це-

лое; следовательно чаще всего встречающийся случай функции,

возвращающей целое значение, не нуждается в описании.

Перед обращением любые фактические аргументы типа FLOAT

преобразуются к типу DOUBLE, любые аргументы типа CHAR или

SHORT преобразуются к типу INT, и, как обычно, имена масси-

вов преобразуются в указатели. Никакие другие преобразования

не выполняются автоматически; в частности, не сравнивает ти-

пы фактических аргументов с типами формальных аргументов.

Если преобразование необходимо, используйте явный перевод

типа (CAST); см. П.п. 15.2, 16.7.

При подготовке к вызову функции делается копия каждого

фактического параметра; таким образом, все передачи аргумен-

тов в языке “C” осуществляются строго по значению. функция

может изменять значения своих формальных параметров, но эти

изменения не влияют на значения фактических параметров. С

другой строны имеется возможность передавать указатель при

таком условии, что функция может изменять значение объекта,

на который этот указатель указывает. Порядок вычисления ар-

гументов в языке не определен; обратите внимание на то, что

различные компиляторы вычисляют по разному.

Допускаются рекурсивные обращения к любой функции.

Первичное выражение, за которым следует точка и иденти-

фикатор, является выражением. Первое выражение должно быть

L-значением, именующим структуру или объединение, а иденти-

фикатор должен быть именем члена структуры или объединения.

Результатом является L-значение, ссылающееся на поименован-

ный член структуры или объединения.

Первичное выражение, за которым следует стрелка (состав-

ленная из знаков - и >) и идентификатор, является выражени-

ем. первое выражение должно быть указателем на структуру или

объединение, а идентификатор должен именовать член этой

структуры или объединения. Результатом является L-значение,

ссылающееся на поименованный член структуры или объединения,

на который указывает указательное выражение.

Следовательно, выражение E1->MOS является тем же самым,

что и выражение (\*E1).MOS. Структуры и объединения рассмат-

риваются в п. 16.5. Приведенные здесь правила использования

структур и объединений не навязываются строго, для того что-

бы иметь возможность обойти механизм типов. См. П. 22.1.

* 194 -

15.2. Унарные операции

Выражение с унарными операциями группируется справо на-

лево.

Унарное-выражение:

* выражение

& L-значение

* выражение

! Выражение

\^ выражение

++ L-значение

* L-значение

L-значение ++

L-значение—

(имя-типа) выражение

SIZEOF выражение

SIZEOF имя-типа

Унарная операция \* означает косвенную адресацию: выраже-

ние должно быть указателем, а результатом является L-значе-

ние, ссылающееся на тот объект, на который указывает выраже-

ние. Если типом выражения является “указатель на...”, то ти-

пом результата будет “...”.

Результатом унарной операции & является указатель на

объект, к которому ссылается L-значение. Если L-значение

имеет тип “...”, то типом результата будет “указатель на

...”.

Результатом унарной операции - (минус) является ее опе-

ранд, взятый с противоположным знаком. Для величины типа

UNSIGNED результат получается вычитанием ее значения из 2\*\*N

(два в степени N), где N-число битов в INT. Унарной операции

+ (плюс) не существует.

Результатом операции логического отрицания ! Является 1,

если значение ее операнда равно 0, и 0, если значение ее

операнда отлично от нуля. Результат имеет тип INT. Эта опе-

рация применима к любому арифметическому типу или указате-

лям.

Операция \^ дает обратный код, или дополнение до едини-

цы, своего операнда. Выполняются обычные арифметические пре-

образования. Операнд должен быть целочисленного типа.

Объект, на который ссылается операнд L-значения префикс-

ной операции ++, увеличивается. значением является новое

значение операнда, но это не L-значение. Выражение ++х экви-

валентно х+=1. Информацию о преобразованиях смотри в разборе

операции сложения (п. 15.4) и операции присваивания (п.

15.14).

Префиксная операция—аналогична префиксной операции

++, но приводит к уменьшению своего операнда L-значения.

При применении постфиксной операции ++ к L-значению ре-

зультатом является значение объекта, на который ссылается

L-значение. После того, как результат принят к сведению,

объект увеличивается точно таким же образом, как и в случае

префиксной операции ++. Результат имеет тот же тип, что и

выражение L-значения.

* 195 -

При применении постфиксной операции—к L-значению ре-

зультатом является значение объекта, на который ссылается

L-значение. После того, как результат принят к сведению,

объект уменьшается точно таким же образом, как и в случае

префиксной операции --. Результат имеет тот же тип, что и

выражение L-значения.

Заключенное в круглые скобки имя типа данных,стоящее пе-

ред выражением , вызывает преобразование значения этого вы-

ражения к указанному типу. Эта конструкция называется пере-

вод (CAST). Имена типов описываются в п. 16.7.

Операция SIZEOF выдает размер своего операнда в байтах.

(Понятие байт в языке не определено, разве только как значе-

ние операции SIZEOF. Однако во всех существующих реализациях

байтом является пространство, необходимое для хранения

объекта типа CHAR). При применении к массиву результатом яв-

ляется полное число байтов в массиве. Размер определяется из

описаний объектов в выражении. Это выражение семантически

является целой константой и может быть использовано в любом

месте, где требуется константа. Основное применение эта опе-

рация находит при связях с процедурами, подобным распредели-

телям памяти, и в системах ввода- вывода.

Операция SIZEOF может быть также применена и к заключен-

ному в круглые скобки имени типа. В этом случае она выдает

размер в байтах объекта указанного типа.

Конструкция SIZEOF (тип) рассматривается как целое, так

что выражение SIZEOF (тип) - 2 эквивалентно выражению

(SIZEOF (тип)9 - 2.

15.3. Мультипликативные операции

Мультипликативные операции \*, /, и % группируются слева

направо. Выполняются обычные арифметические преобразования.

Мультипликативное-выражение:

выражение \* выражение

выражение / выражение

выражение % выражение

Бинарная операция \* означает умножение. Операция \* ассо-

циативна, и выражения с несколькими умножениями на одном и

том же уровне могут быть перегруппированы компилятором.

Бинарная операция / означает деление. При делении поло-

жительных целых осуществляется усечение по направлению к ну-

лю, но если один из операндов отрицателен, то форма усечения

зависит от используемой машины. На всех машинах, охватывае-

мых настоящим руководством, остаток имеет тот же знак , что

и делимое. Всегда справедливо, что (A/B)\*B+A%B равно A (если

B не равно 0).

Бинарная операция % выдает остаток от деления первого

выражения на второе. Выполняются обычные арифметические пре-

образования. Операнды не должны быть типа FLOAT.

15.4. Аддитивные операции

Аддитивные операции + и - группируются слева направо.

выполняются обычные арифметические преобразования. Для каж-

дой операции имеются некоторые дополнительные возможности,

связанные с типами операндов.

* 196 -

Аддитивное-выражение:

выражение + выражение

выражение - выражение

Результатом операции + является сумма операндов. Можно скла-

дывать указатель на объект в массиве и значение любого цело-

численного типа. во всех случаях последнее преобразуется в

адресное смещение посредством умножения его на длину объек-

та, на который указывает этот указатель. Результатом являет-

ся указатель того же самого типа, что и исходный указатель,

который указывает на другой объект в том же массиве, смещен-

ный соответствующим образом относительно первоначального

объекта. Таким образом, если P является указателем объекта в

массиве, то выражение P+1 является указателем на следующий

объект в этом массиве.

Никакие другие комбинации типов для указателей не разре-

шаются.

Операция + ассоциативна, и выражение с несколькими сло-

жениями на том же самом уровне могут быть переупорядочены

компилятором.

Результатом операции - является разность операндов. Вы-

полняются обычные арифметические преобразования. Кроме того,

из указателя может быть вычтено значение любого целочислен-

ного типа, причем, проводятся те же самые преобразования,

что и при операции сложения.

Если вычитаются два указателя на объекты одинакового ти-

па, то результат преобразуется (делением на длину объекта) к

типу INT, представляя собой число объектов, разделяющих ука-

зываемые объекты. Если эти указатели не на объекты из одного

и того же массива, то такое преобразование, вообще говоря,

даст неожиданные результаты, потому что даже указатели на

объекты одинакового типа не обязаны отличаться на величину,

кратную длине объекта.

15.5. Операции сдвига

Операции сдвига << и >> группируются слева направо. Для

обеих операций проводятся обычные арифметические преобразо-

вания их операндов, каждый из которых должен быть целочис-

ленного типа. Затем правый операнд преобразуется к типу INT;

результат имеет тип левого операнда. Результат не определен,

если правый операнд отрицателен или больше или равен, чем

длина объекта в битах.

Выражение-сдвига:

выражение << выражение

выражение >> выражение

Значением выражения E1<<E2 является E1 (интерпретируемое как

комбинация битов), сдвинутое влево на E2 битов; освобождаю-

щиеся биты заполняются нулем. значением выражения E1>>E2 яв-

ляется E1, сдвинутое вправо на E2 битовых позиций. Если E1

имеет тип UNSIGNE, то сдвиг вправо гарантированно будет ло-

гическим (заполнение нулем); в противном случае сдвиг может

быть (и так и есть на PDP-11) арифметическим (освобождающие-

ся биты заполняются копией знакового бита).

* 197 -

15.6. Операции отношения

Операции отношения группируются слева направо, но этот

факт не очень полезен; выражение A<B<C не означает того, что

оно казалось бы должно означать.

Выражение-отношения:

выражение < выражение

выражение > выражение

выражение <= выражение

выражение >= выражение

Операции < (меньше), > (больше), <= (меньше или равно) и >=

(больше или равно) все дают 0, если указанное отношение лож-

но, и 1, если оно истинно. Результат имеет тип ITN. Выполня-

ются обычные арифметические преобразования. Могут сравни-

ваться два указателя; результат зависит от относительного

расположения указываемых объектов в адресном пространстве.

Сравнение указателей переносимо только в том случае, если

указатели указывают на объекты из одного и того же массива.

15.7. Операции равенства Выражение-равенства:

выражение == выражение

выражение != выражение

Операции == (равно) и != (не равно) в точности аналогичны

операциям отношения, за исключением того, что они имеют бо-

лее низкий уровень старшинства. (Поэтому значение выражения

A<B==C<D равно 1 всякий раз, когда выражение A<B и C<D имеют

одинаковое значение истинности).

Указатель можно сравнивать с целым, но результат будет

машинно- независимым только в том случае, если целым являет-

ся константа 0. Гарантируется, что указатель, которому прис-

воено значение 0, не указывает ни на какой объект и на самом

деле оказывается равным 0; общепринято считать такой указа-

тель нулем.

15.8. Побитовая операция 'и' Выражение-и:

выражение & выражение

Операция & является ассоциативной, и включающие & выражения

могут быть переупорядочены. Выполняются обычные арифметичес-

кие преобразования; результатом является побитовая функция

'и' операндов. Эта операция применима только к операндам це-

лочисленного типа.

15.9. Побитовая операция исключающего 'или' Выражение-исключающего-или:

выражение ^ выражение

Операция ^ является ассоциативной, и включающие ^ выражения

могут быть переупорядочены. выполняются обычные арифметичес-

кие преобразования; результатом является побитовая функция

исключающего 'или' операндов. Операция применима только к

операндам целочисленного типа.

* 198 -

15.10. Побитовая операция включающего 'или' Выражение-включающего-или:

выражение \! Выражение

Операция \! Является ассоциативной, и содержащие \! Выраже-

ния могут быть переупорядочены. выполняются обычные арифме-

тические преобразования; результатом является побитовая фун-

кция включающего 'или' операндов. Операция применима только

к операндам целочисленного типа.

15.11. Логическая операция 'и' Выражение-логического-и:

выражение && выражение

Операция && группируется слева направо. Она возвращает 1,

если оба ее операнда отличны от нуля, и 0 в противном слу-

чае. В отличие от & операция && гарантирует вычисление слева

направо; более того, если первый операнд равен 0, то значе-

ние второго операнда вообще не вычисляется.

Операнды не обязаны быть одинакового типа, но каждый из

них должен быть либо одного из основных типов, либо указате-

лем. результат всегда имеет тип ITN.

15.12. Операция логического 'или' Выражение-логического-или:

выражение \!\! выражение

Операция \!\! Группируется слева направо. Она возвращает 1,

если один из операндов отличен от нуля, и 0 в противном слу-

чае. В отличие от операции \! Операция \!\! Гарантирует вы-

числение слева направо; более того, если первый операнд от-

личен от нуля, то значение второго операнда вообще не вычис-

ляется.

Операнды не обязаны быть одинакового типа, но каждый из

них должен быть либо одного из основных типов, либо указате-

лем. Результат всегда имеет тип INT.

15.13. Условная операция Условное-выражение:

выражение ? выражение : выражение

Условные выражения группируются слево направо. Вычисляется

значение первого выражения, и если оно отлично от нуля, то

результатом будет значение второго выражения; в противном

случае результатом будет значение третьего выражения. Если

это возможно, проводятся обычные арифметические преобразова-

ния, с тем, чтобы привести второе и третье выражения к обще-

му типу; в противном случае, если оба выражения являются

указателями одинакового типа, то результат имеет тот же тип;

в противном случае одно выражение должно быть указателем, а

другое - константой 0, и результат будет иметь тип указате-

ля. Вычисляется только одно из второго и третьего выражений.

* 199 -

15.14. Операция присваивания

Имеется ряд операций присваивания, каждая из которых

группируется слева направо. Все операции требуют в качестве

своего левого операнда L-значение, а типом выражения присва-

ивания является тип его левого операнда. Значением выражения

присваивания является значение, хранимое в левом операнде

после того, как присваивание уже будет произведено. Две час-

ти составной операции присваивания являются отдельными лек-

семами.

Выражение-присваивания:

L-значение = выражение

L-значение += выражение

L-значение -= выражение

L-значение \*= выражение

L-значение /= выражение

L-значение %= выражение

L-значение >>= выражение

L-значение <<= выражение

L-значение &= выражение

L-значение ^= выражение

L-значение \!= выражение

Когда производится простое присваивание C'=', значение

выражения заменяет значение объекта, на которое ссылается

L-значение. Если оба операнда имеют арифметический тип, то

перед присваиванием правый операнд преобразуется к типу ле-

вого операнда.

О свойствах выражения вида E1 оп = E2, где Oп - одна из

перечисленных выше операций, можно сделать вывод, если

учесть, что оно эквивалентно выражению E1 = E1 оп (E2); од-

нако выражение E1 вычисляется только один раз. В случае опе-

раций += и -= левый операнд может быть указателем, причем

при этом (целочисленный) правый операнд преобразуется таким

образом, как объяснено в п. 15.4; все правые операнды и все

отличные от указателей левые операнды должны иметь арифмети-

ческий тип.

Используемые в настоящее время компиляторы допускают

присваивание указателя целому, целого указателю и указателя

указателю другого типа. такое присваивание является чистым

копированием без каких-либо преобразований. Такое употребле-

ние операций присваивания является непереносимым и может

приводить к указателям, которые при использовании вызывают

ошибки адресации. Тем не менее гарантируется, что присваива-

ние указателю константы 0 дает нулевой указатель, который

можно отличать от указателя на любой объект.

15.15. Операция запятая Выражение-с-запятой:

выражение , выражение

* 200 -

Пара выражений, разделенных запятой, вычисляется слева нап-

раво и значение левого выражения отбрасывается. Типом и зна-

чением результата является тип и значение правого операнда.

Эта операция группируется слева направо. В контексте, где

запятая имеет специальное значение, как, например, в списке

фактических аргументов функций (п. 15.1) Или в списках ини-

циализаторов (п. 16.6), Операция запятая, описываемая в этом

разделе, может появляться только в круглых скобках; напри-

мер, функция

F(A,(T=3,T+2),C)

имеет три аргумента, второй из которых имеет значение 5.

16. Описания

Описания используются для указания интерпретации, кото-

рую язык “C” будет давать каждому идентификатору; они не

обязательно резервируют память, соответствующую идентифика-

тору. Описания имеют форму

Описание:

спецификаторы-описания список-описателей

необ;

Описатели в списке описателей содержат описываемые идентифи-

каторы. Спецификаторы описания представляют собой последова-

тельность спецификаторов типа и спецификаторов класса памя-

ти.

Спецификаторы-описания:

спецификатор-типа спецификаторы-описания

необ

спецификатор-класса-памяти спецификатор-описания

необ

список должен быть самосогласованным в смысле, описываемом

ниже.

16.1. Спецификаторы класса памяти Ниже перечисляются спецификаторы класса памяти:

Спецификатор-класса-памяти:

AUTO

STATIC

EXTERN

REGISTER

TYPEDEF

Спецификатор TYPEDEF не реализует памяти и называется

“спецификатором класса памяти” только по синтаксическим со-

ображениям; это обсуждается в п. 16.8. Смысл различных клас-

сов памяти был обсужден в п. 12.

Описания AUTO, STATIC и REGISTER служат также в качестве

определений в том смысле, что они вызывают резервирование

нужного количества памяти. В случае EXTERN должно присутст-

вовать внешнее определение (п. 18) Указываемых идентификато-

ров где-то вне функции, в которой они описаны.

* 201 -

Описание REGISTER лучше всего представлять себе как опи-

сание AUTO вместе с намеком компилятору, что описанные таким

образом переменные будут часто использоваться. Эффективны

только несколько первых таких описаний. Кроме того, в регис-

трах могут храниться только переменные определенных типов;

на PDP-11 это INT, CHAR или указатель. Существует и другое

ограничение на использование регистровых переменных: к ним

нельзя применять операцию взятия адреса &. При разумном ис-

пользовании регистровых описаний можно ожидать получения

меньших по размеру и более быстрых программ, но улучшение в

будущем генерирования кодов может сделать их ненужными.

Описание может содержать не более одного спецификатора

класса памяти. Если описание не содержит спецификатора клас-

са памяти, то считается, что он имеет значение AUTO, если

описание находится внутри некоторой функции, и EXTERN в про-

тивном случае. исключение: функции никогда не бывает автома-

тическими.

16.2. Спецификаторы типа

Ниже перечисляются спецификаторы типа.

Спецификатор-типа:

CHAR

SHORT

INT

LONG

UNSIGNED

FLOAT

DOUBLE

спецификатор-структуры-или-объединения

определяющее-тип-имя

Слова LONG, SHORT и USIGNED можно рассматривать как при-

лагательные; допустимы следующие комбинации:

SHORT INT

LONG INT

USIGNED INT

LONG FLOAT

Последняя комбинация означает то же, что и DOUBLE. В осталь-

ном описание может содержать не более одного спецификатора

типа. Если описание не содержит спецификатора типа, то счи-

тается, что он имеет значение INT.

Спецификаторы структур и объединений обсуждаются в п.

16.5; Описания с определяющими тип именами TYPEDEF обсужда-

ются в п. 16.8.

* 202 -

16.3. Описатели

Входящий в описание список описателей представляет собой

последовательность разделенных запятыми описателей, каждый

из которых может иметь инициализатор.

Список-описателей:

инициализируемый-описатель

инициализируемый-описатель, список-описателей

инициализируемый-описатель:

описатель-инициализатор

необ

Инициализаторы описываются в п. 16.6. Спецификаторы и описа-

ния указывают тип и класс памяти объектов, на которые ссыла-

ются описатели. Описатели имеют следующий синтаксис:

описатель:

идентификатор

( описатель )

* описатель описатель () описатель [константное-выражение

необ]

Группирование такое же как и в выражениях.

16.4. Смысл описателей

Каждый описатель рассматривается как утверждение того,

что когда конструкция той же самой формы, что и описатель,

появляется в выражении, то она выдает объект указанного типа

и указанного класса памяти. Каждый описатель содержит ровно

один идентификатор; это именно тот идентификатор, который и

описывается.

Если в качестве описателя появляется просто идентифика-

тор, то он имеет тип, указываемый в специфицирующем заголов-

ке описания.

Описатель в круглых скобках идентичен описателю без

круглых скобок, но круглые скобки могут изменять связи в

составных описателях. Примеры смотри ниже.

Представим себе описание

T DI

где T - спецификатор типа (подобный INT и т.д.), а DI - опи-

сатель. Предположим, что это описание приводит к тому, что

соответствующий идентификатор имеет тип “...T”, где “...”

пусто, если DI просто отдельный идентификатор (так что тип X

в “INT X” просто INT). Тогда , если DI имеет форму

\*D

то содержащийся идентификатор будет иметь тип “... Указатель

на T”.

* 203 -

Если DI имеет форму

D()

то содержащийся идентификатор имеет тип “... Функция, возв-

ращающая T”.

Если DI имеет форму

D[константное-выражение]

или

D[ ]

то содержащийся идентификатор имеет тип “...массив T”. В

первом случае константным выражением является выражение,

значение которого можно определить во время компиляции и ко-

торое имеет тип INT. (Точное определение константного выра-

жения дано в п. 23). Когда несколько спецификаций вида “мас-

сив из” оказываются примыкающими, то создается многомерный

массив; константное выражение, задающее границы массивов,

может отсутствовать только у первого члена этой последова-

тельности. Такое опускание полезно, когда массив является

внешним и его фактическое определение, которое выделяет па-

мять, приводится в другом месте. Первое константное выраже-

ние может быть опущено также тогда, когда за описателем сле-

дует инициализация. В этом случае размер определяется по

числу приведенных инициализируемых элементов.

Массив может быть образован из элементов одного из ос-

новных типов, из указателей, из структур или объединений или

из других массивов (чтобы образовать многомерный массив).

Не все возможности, которые разрешены с точки зрения

указанного выше синтаксиса, фактически допустимы. Имеются

следующие ограничения: функции не могут возвращать массивы,

структуры, объединения или функции, хотя они могут возвра-

щать указатели на такие вещи; не существует массивов функ-

ций, хотя могут быть массивы указателей на функции. Анало-

гично, структуры или объединения не могут содержать функцию,

но они могут содержать указатель на функцию.

В качестве примера рассмотрим описание

INT I, \*IP, F(), \*FIP(), (\*PFI)();

в котором описывается целое I, указатель IP на целое, функ-

ция F, возвращающая целое, функция FIP, возвращающая указа-

тель на целое, и указатель PFI на функцию, которая возвраща-

ет целое. Особенно полезно сравнить два последних описателя.

Связь в \*FIP() можно представить в виде \*(FIP()), так что

описанием предполагается, а такой же конструкцией в выраже-

нии требуется обращение к функции FIP и последующее исполь-

зование косвенной адресации для выдачи с помощью полученного

результата (указателя) целого. В описателе (\*PFI)() дополни-

* 204 -

тельные скобки необходимы, поскольку они точно так же, как и

в выражении, указывают, что косвенная адресация через указа-

тель на функцию выдает функцию, которая затем вызывается;

эта вызванная функция возвращает целое.

В качестве другого примера приведем описание

FLOAT FA[17], \*AFP[17];

в котором описывается массив чисел типа FLOAT и массив ука-

зателей на числа типа FLOAT. Наконец,

STATIC INT X3D[3][5][7];

описывает статический трехмерный массив целых размером

3\*5\*7. более подробно, X3D является массивом из трех элемен-

тов; каждый элемент является массивом пяти массивов; каждый

последний массив является массивом из семи целых. Каждое из

выражений X3D, X3D[I], X3D[I][J] и X3D[I][J][K] может разум-

ным образом появляться в выражениях. Первые три имеют тип

“массив”, последнее имеет тип INT.

16.5. Описание структур и объединений

Структура - это объект, состоящий из последовательности

именованных членов. каждый член может быть произвольного ти-

па. Объединение - это объект, который в данный момент может

содержать любой из нескольких членов. Спецификаторы и

объединения имеют одинаковую форму.

Спецификатор-структуры-или-объединения

структура-или-объединение \( список-описаний-структуры\)

идентификатор структуры-или-объединения

\(список-описаний-структуры\)

идентификатор структуры-или-объединения

Структура-или-объединение:

STRUCT

UNION

Список-описаний-структуры является последовательностью опи-

саний членов структуры или объединения:

Список-описаний-структуры:

описание-структуры

описание-структуры список-описаний-структуры

описание-структуры:

спецификатор-типа список-описателей-структуры

список-описателей-структуры:

описатель-структуры

описатель-структуры, список-описателей-структуры

В обычном случае описатель структуры является просто описа-

телем члена структуры или объединения. Член структуры может

также состоять из специфицированного числа битов. Такой член

называется также полем; его длина отделяется от имени поля

двоеточием.

* 205 -

Описатель-структуры:

описатель

описатель: константное выражение

: константное выражение

Внутри структуры описанные в ней объекты имеют адреса, кото-

рые увеличиваются в соответствии с чтением их описаний слева

направо. Каждый член структуры, который не является полем,

начинается с адресной границы, соответствующей его типу;

следовательно в структуре могут оказаться неименованные ды-

ры. Члены, являющиеся полями, помещаются в машинные целые;

они не перекрывают границы слова. Поле, которое не умещается

в оставшемся в данном слове пространстве, помещается в сле-

дующее слово. Поля выделяются справа налево на PDP-11 и сле-

ва направо на других машинах.

Описатель структуры, который не содержит описателя, а

только двоеточие и ширину, указывает неименованное поле, по-

лезное для заполнения свободного пространства с целью соот-

ветствия задаваемых извне схемам. Специальный случай неиме-

нованного поля с шириной 0 используется для указания о вы-

равнивании следующего поля на границу слова. При этом пред-

полагается, что “следующее поле” действиетльно является по-

лем, а не обычным членом структуры, поскольку в последнем

случае выравнивание осуществляется автоматически.

Сам язык не накладывает ограничений на типы объектов,

описанных как поля, но от реализаций не требуется обеспечи-

вать что-либо отличное от целых полей. Более того, даже поля

типа INT могут рассматриваться как неимеющие знака. На

PDP-11 поля не имеют знака и могут принимать только целые

значения. Во всех реализациях отсутствуют массивы полей и к

полям не применима операция взятия адреса &, так что не су-

ществует и указателей на поля.

Объединение можно представить себе как структуру, все

члены которой начинаются со смещения 0 и размер которой дос-

таточен, чтобы содержать любой из ее членов. В каждый момент

объединение может содержать не более одного из своих членов.

Спецификатор структуры или объединения во второй форме,

т.е. Один из

STRUCT идентификатор \(список-описаний-структуры\)

UNION идентификатор \(список-описаний-структуры\)

описывает идентификатор в качестве ярлыка структуры (или яр-

лыка объединения) структуры, специфицированной этим списком.

Последующее описание может затем использовать третью форму

спецификатора, один из

STRUCT идентификатор

UNION идентификатор

* 206 -

Ярлыки структур дают возможность определения структур, кото-

рые ссылаются на самих себя; они также позволяют неоднократ-

но использовать приведенную только один раз длинную часть

описания. Запрещается описывать структуру или объединение,

которые содержат образец самого себя, но структура или

объединение могут содержать указатель на структуру или

объединение такого же вида, как они сами.

Имена членов и ярлыков могут совпадать с именами обычных

переменных. Однако имена ярлыков и членов должны быть взаим-

но различными.

Две структуры могут иметь общую начальную последователь-

ность членов; это означает, что тот же самый член может поя-

виться в двух различных структурах, если он имеет одинаковый

тип в обеих структурах и если все предыдущие члены обеих

структур одинаковы. (Фактически компилятор только проверяет,

что имя в двух различных структурах имеет одинаковый тип и

одинаковое смещение, но если предшествующие члены отличают-

ся, то конструкция оказывается непереносимой).

Вот простой пример описания структуры:

STRUCT TNODE \( CHAR TWORD[20];

INT COUNT;

STRUCT TNODE \*LEFT;

STRUCT TNODE \*RIGHT;

\);

Такая структура содержит массив из 20 символов, целое и два

указателя на подобные структуры. Как только приведено такое

описание, описание

STRUCT TNODE S, \*SP;

говорит о том, что S является структурой указанного вида, а

SP является указателем на структуру указанного вида. При на-

личии этих описаний выражение

SP->COUNT

ссылается к полю COUNT структуры, на которую указывает SP;

выражение

S.LEFT

ссылается на указатель левого поддерева в структуре S, а вы-

ражение

S.RIGHT->TWORD[0]

ссылается на первый символ члена TWORD правого поддерева из

S.

* 207 -

16.6. Инициализация

Описатель может указывать начальное значение описываемо-

го идентификатора. Инициализатор состоит из выражения или

заключенного в фигурные скобки списка значений, перед кото-

рыми ставится знак =.

Инициализатор:

= выражение

= \(список-инициализатора\)

= \(список-инициализатора,\)

список-инициализатора:

выражение

список-инициализатора,список-инициализатора

\(список-инициализатора\)

Все выражения, входящие в инициализатор статической или

внешней переменной, должны быть либо константными выражения-

ми, описываемыми в п. 23, Либо выражениями, которые сводятся

к адресу ранее описанной переменной, возможно смещенному на

константное выражение. Автоматические и регистровые перемен-

ные могут быть инициализированы произвольными выражениями,

включающими константы и ранее описанные переменные и функ-

ции.

Гарантируется, что неинициализированные статические и

внешние переменные получают в качестве начальных значений

0;неинициализированные автоматические и регистровые перемен-

ные в качестве начальных значений содержат мусор.

Когда инициализатор применяется к скаляру (указателю или

объекту арифметического типа), то он состоит из одного выра-

жения, возможно заключенного в фигурные скобки. Начальное

значение объекта находится из выражения; выполняются те же

самые преобразования, что и при присваивании.

Когда описываемая переменная является агрегатом (струк-

турой или массивом ), то инициализатор состоит из заключен-

ного в фигурные скобки и разделенного запятыми списка иници-

ализаторов для членов агрегата. Этот список составляется в

порядке возрастания индекса или в соответствии с порядком

членов. Если агрегат содержит подагрегаты, то это правило

применяется рекурсивно к членам агрегата. Если количество

инициализаторов в списке оказывается меньше числа членов аг-

регата, то оставшиеся члены агрегата заполняются нулями.

Запрещается инициализировать объединения или автоматические

агрегаты.

Фигурные скобки могут быть опущены следующим образом.

Если инициализатор начинается с левой фигурной скобки, то

последующий разделенный запятыми список инициализаторов ини-

циализирует члены агрегата; будет ошибкой, если в списке

окажется больше инициализаторов, чем членов агрегата. Если

однако инициализатор не начинается с левой фигурной скобки,

то из списка берется только нужное для членов данного агре-

гата число элементов; оставшиеся элементы используются для

инициализации следующего члена агрегата, частью которого яв-

ляется настоящий агрегат.

* 208 -

Последнее сокращение допускает возможность инициализации

массива типа CHAR с помощью строки. В этом случае члены мас-

сива последовательно инициализируются символами строки.

Например,

INT X[] = \(1,3,5\);

описывает и инициализирует X как одномерный массив; посколь-

ку размер массива не специфицирован, а список инициализитора

содержит три элемента, считается, что массив состоит из трех

членов.

Вот пример инициализации с полным использованием фигур-

ных скобок:

FLOAT \*Y[4][3] = \(

( 1, 3, 5 ),

( 2, 4, 6 ),

( 3, 5, 7 ),

\);

Здесь 1, 3 и 5 инициализируют первую строку массива Y[0], а

именно Y[0][0], Y[0][1] и Y[0][2]. Аналогичным образом сле-

дующие две строчки инициализируют Y[1] и Y[2]. Инициализатор

заканчивается преждевременно, и, следовательно массив Y[3]

инициализируется нулями. В точности такого же эффекта можно

было бы достичь, написав

FLOAT Y[4][3] = \(

1, 3, 5, 2, 4, 6, 3, 5, 7

\);

Инициализатор для Y начинается с левой фигурной скобки, но

инициализатора для Y[0] нет. Поэтому используется 3 элемента

из списка. Аналогично следующие три элемента используются

последовательно для Y[1] и Y[2]. следующее описание

FLOAT Y[4][3] = \(

(1), (2), (3), (4)

\);

инициализирует первый столбец Y (если его рассматривать как

двумерный массив), а остальные элементы заполняются нулями.

И наконец, описание

CHAR MSG[] = “SYNTAX ERROR ON LINE %S\N”;

демонстрирует инициализацию элементов символьного массива с

помощью строки.

16.7. Имена типов

В двух случаях (для явного указания типа преобразования

в конструкции перевода и для аргументов операции SIZEOF) же-

лательно иметь возможность задавать имя типа данных. Это

осуществляется с помощью “имени типа”, которое по существу

является описанием объекта такого типа , в котором опущено

имя самого объекта.

* 209 -

Имя типа:

спецификатор-типа абстрактный-описатель

абстрактный-описатель:

пусто

(абстрактный-описатель)

* абстрактный описатель абстрактный-описатель () абстрактный-описатель [константное выражение

необ]

Во избежании двусмысленности в конструкции

(абстрактный описатель)

требуется, чтобы абстрактный-описатель был непуст. При этом

ограничении возможно однозначено определить то место в абст-

рактном-описателе, где бы появился идентификатор, если бы

эта конструкция была описателем в описании. Именованный тип

совпадает тогда с типом гипотетического идентификатора. Нап-

ример, имена типов

INT

INT \*

INT \*[3]

INT (\*)[3]

INT \*()

INT (\*)()

именуют соответственно типы “целый”, “указатель на целое”,

“массив из трех указателей на целое”, “указатель на массив

из трех целых”, “ функция, возвращающая указатель на целое”

и “указатель на функцию, возвращающую целое”.

16.8. TYPEDEF

Описания, в которых “класс памяти”специфицирован как

TYPEDEF, не вызывают выделения памяти. вместо этого они оп-

ределяют идентификаторы ,которые позднее можно использовать

так, словно они являются ключевыми словами, имеющими основ-

ные или производные типы.

Определяющее-тип-имя

идентификатор

В пределах области действия описания со спецификатором

TYPEDEF каждый идентификатор, являющийся частью любого опи-

сателя в этом описании, становится синтаксически эквивалент-

ным ключевому слову, имеющему тот тип , который ассоциирует

с идентификатором в описанном в п. 16.4 Смысле. Например,

после описаний

TYPEDEF INT MILES, >KLICKSP;

TYPEDEF STRUCT ( DOUBLE RE, IM; ) COMPLEX;

конструкции

MILES DISTANCE;

EXTERN KLICKSP METRICP;

COMPLEX Z, \*ZP;

* 210 -

становятся законными описаниями; при этом типом DISTANCE яв-

ляется INT, типом METRICP - “указатель на INT”, типом Z -

специфицированная структура и типом ZP - указатель на такую

структуру.

Спецификатор TYPEDEF не вводит каких-либо совершенно но-

вых типов, а только определяет синонимы для типов, которые

можно было бы специфицировать и другим способом. Так в при-

веденном выше примере переменная DISTANCE считается имеющей

точно такой же тип, что и любой другой объект, описанный в

INT.

17. Операторы

За исключением особо оговариваемых случаев, операторы

выполняются последовательно.

17.1. Операторное выражение

Большинство операторов являются операторными выражения-

ми, которые имеют форму

выражение;

обычно операторные выражения являются присваиваниями или об-

ращениями к функциям.

17.2. Составной оператор (или блок)

С тем чтобы допустить возможность использования несколь-

ких операторов там, где ожидается присутствие только одного,

предусматривается составной оператор (который также и экви-

валентно называют “блоком”):

составной оператор:

\(список-описаний список-операторов

необ необ\)

список-описаний:

описание

описание список-описаний

список-операторов:

оператор

оператор список-операторов

Если какой-либо идентификатор из списка-описаний был описан

ранее, то во время выполнения блока внешнее описание подав-

ляется и снова вступает в силу после выхода из блока.

Любая инициализация автоматических и регистрационных пе-

ременных проводится при каждом входе в блок через его нача-

ло. В настоящее время разрешается (но это плохая практика)

передавать управление внутрь блока; в таком случае эти ини-

циализации не выполняются. Инициализации статических пере-

менных проводятся только один раз, когда начинается выполне-

ние программы.

Находящиеся внутри блока внешние описания не

резервируют памяти, так что их инициализация не

разрешается.

* 211 -

17.3. Условные операторы Имеются две формы условных операторов:

IF (выражение) оператор

IF (выражение) оператор ELSE оператор

В обоих случаях вычасляется выражение и, если оно отлично от

нуля, то выполняется первый подоператор. Во втором случае,

если выражение равно нулю, выпалняется второй подоператор.

Как обычно, двусмысленность “ELSE” разрешается связываением

ELSE с последним встречающимся IF, у которого нет ELSE.

17.4. Оператор WHILE

Оператор WHILE имеет форму

WHILE (выражение) оператор

Подоператор выполняется повторно до тех пор, пока значение

выражения остается отличным от нуля. проверка производится

перед каждым выполнением оператора.

17.5. Оператор DO

Оператор DO имеет форму

DO оператор WHILE (выражения)

Оператор выполняется повторно до тех пор, пока значение выражения не станет равным нулю. Проверка производится после каждого выполнения оператора.

17.6. Оператор FOR

Оператор FOR имеет форму

(выражение-1 ; выражение-2 ; выражение-3 )оператор

необ необ необ

Оператор FOR эквивалентен следующему

выражение-1;

WHILE (выражение-2) \(

оператор

выражение-3

\)

Таким образом, первое выражение определяет инициализацию

цикла; второе специфиуирует проверку, выполняемую перед каж-

дой итерацией, так что выход из цикла происходит тогда, ког-

да значение выражения становится нулем; третье выражение

часто задает приращение параметра, которое проводится после

каждой итерации.

Любое выражение или даже все они могут быть опущены. Ес-

ли отсутствует второе выражение, то предложение с WHILE счи-

тается эквивалентным WHILE(1); другие отсутствующие выраже-

ния просто опускаются из приведенного выше расширения.

* 212 -

17.7. Оператор SWITCH

Оператор SWITCH (переключатель), вызывает передачу уп-

равления к одному из нескольких операторов, в зависимости от

значения выражения. Оператор имеет форму

SWITCH (выражение) оператор

В выражении проводятся обычные арифметические преобразова-

ния, но результат должен иметь тип INT. Оператор обычно яв-

ляется составным. Любой оператор внутри этого оператора мо-

жет быть помечен одним или более вариантным префиксом CASE,

имеющим форму:

CASE констанстное выражение:

где константное выражение должно иметь тип INT. Никакие две

вариантные константы в одном и том же переключателе не могут

иметь одинаковое значение. точное определение константного

выражения приводится в п. 23.

Кроме того, может присутствовать самое большее один опе-

раторный префикс вида

DEFAULT:

При выполнении оператора SWITCH вычисляется входящее в

него выражение и сравнивается с каждой вариантной констан-

той. Если одна из вариантных констант оказывается равной

значению этого выражения, то управление передается операто-

ру, который следует за совпадающим вариантным префиксом. Ес-

ли ни одна из вариантных констант не совпадает со значением

выражения и если при этом присутствует префикс DEFAULT, то

управление передается оператору, помеченному этим префиксом.

если ни один из вариантов не подходит и префикс DEFAULT от-

сутствует, то ни один из операторов в переключателе не вы-

полняется.

Сами по себе префиксы CASE и DEFAULT не изменяют поток

управления, которое беспрепятсвенно проходит через такие

префиксы. Для выхода из переключателя смотрите оператор

BREAK, п. 17.8.

Обычно оператор, который входит в переключатель, являет-

ся составным. Описания могут появляться в начале этого опе-

ратора, но инициализации автоматических и регистровых пере-

менных будут неэффективными.

17.8. Оператор BREAK

Оператор

BREAK;

вызывает завершение выполнения наименьшего охватывающего

этот оператор оператора WHILE, DO, FOR или SWITCH; управле-

ние передается оператору, следующему за завершенным операто-

ром.

* 213 -

17.9. Оператор CONTINUE

Оператор

CONTINUE;

приводит к передаче управления на продолжающую цикл часть

наименьшего охватывающего этот оператор оператора WHILE, DO

или FOR; то есть на конец цикла. Более точно, в каждом из

операторов

WHILE(...) \( DO \( FOR(...) \(

... ... ...

CONTIN: ; CONTIN: ; CONTIN: ;

\) \) WHILE(...); \)

Оператор CONTINUE эквивалентен оператору GOTO CONTIN. (За

CONTIN: следует пустой оператор; см. П. 17.13.).

17.10. Оператор возврата

Возвращение из функции в вызывающую программу осуществ-

ляется с помощью оператора RETURN, который имеет одну из

следующих форм

RETURN;

RETURN выражение;

В первом случае возвращаемое значение неопределено. Во вто-

ром случае в вызывающую функцию возвращается значение выра-

жения. Если требуется, выражение преобразуется к типу функ-

ции, в которой оно появляется, как в случае присваивания.

Попадание на конец функции эквивалентно возврату без возвра-

щаемого значения.

17.11. Оператор GOTO

Управление можно передавать безусловно с помощью опера-

тора

GOTO идентификатор1

идентификатор должен быть меткой (п. 9.12), Локализованной в

данной функции.

17.12. Помеченный оператор

Перед любым оператором может стоять помеченный префикс

вида

идентификатор:

который служит для описания идентификатора в качестве метки.

Метки используются только для указания места, куда передает-

ся управление оператором GOTO. Областью действия метки явля-

ется данная функция, за исключением любых подблоков, в кото-

рых тот же идентификатор описан снова. Смотри п. 19.

* 214 -

17.13. Пустой оператор Пустой оператор имеет форму:

;

Пустой оператор оказывается полезным, так как он позволяет

поставить метку перед закрывающей скобкой \) составного опе-

ратора или указать пустое тело в операторах цикла, таких как

WHILE.

18. Внешние определения

C-программа представляет собой последовательность внеш-

них определений. Внешнее определение описывает идентификатор

как имеющий класс памяти EXTERN (по умолчанию), или возможно

STATIC, и специфицированный тип. Спецификатор типа (п. 16.2)

Также может быть пустым; в этом случае считается, что тип

является типом INT. Область действия внешних определений

распространяется до конца файла, в котором они приведены,

точно так же , как влияние описаний простирается до конца

блока. Синтаксис внешних определений не отличается от син-

таксиса описаний, за исключением того, что только на этом

уровне можно приводить текст функций.

18.1. Внешнее определение функции

Определение функции имеет форму

определение-функции:

спецификаторы-описания описатель-функции

тело-функции

необ

Единственными спецификаторами класса памяти, допускаемыми в

качестве спецификаторов-описания, являются EXTERN или

STATIC; о различии между ними смотри п. 19.2. Описатель фун-

кции подобен описателю для “функции, возвращающей...”, за

исключением того, что он перечисляет формальные параметры

определяемой функции.

Оисатель-функции:

описатель (список-параметров

необ)

список параметров:

идентификатор

идентификатор, список-параметров

тело-функции имеет форму

тело-функции:

список-описаний составной-оператор

* 215 -

Идентификаторы из списка параметров и только они могут

быть описаны в списке описаний. Любой идентификатор, тип ко-

торого не указан, считается имеющим тип INT. Единственным

допустимым здесь спецификатором класса памяти является

REGISTER; если такой класс памяти специфицирован, то в нача-

ле выполнения функции соответствующий фактический параметр

копируется, если это возможно, в регистр.

Вот простой пример полного определения функции:

INT MAX(A, B, C)

INT A, B, C;

\(

INT M;

M = (A>B) ? A:B;

RETURN((M>C) ? M:C);

\)

Здесь INT - спецификатор-типа, MAX(A,B,C) - описатель-функ-

ции, INT A,B,C; - список-описаний формальных параметров, \(

... \) - Блок, содержащий текст оператора.

В языке “C” все фактические параметры типа FLOAT преоб-

разуются к типу DOUBLE, так что описания формальных парамет-

ров, объявленных как FLOAT, приспособлены прочесть параметры

типа DOUBLE. Аналогично, поскольку ссылка на массив в любом

контексте (в частности в фактическом параметре) рассматрива-

ется как указатель на первый элемент массива, описания фор-

мальных параметров вила “массив ...” приспособлены прочесть

: “указатель на ...”. И наконец, поскольку структуры,

объединения и функции не могут быть переданы функции, бесс-

мысленно описывать формальный параметр как структуру,

объединение или функцию (указатели на такие объекты, конеч-

но, допускаются).

18.2. Внешние определения данных

Внешнее определение данных имеет форму

определение-данных:

описание

Классом памяти таких данных может быть EXTERN (в частности,

по умолчанию) или STATIC, но не AUTO или REGISTER.

19. Правила, определяющие область действия

Вся C-программа необязательно компилируется одновремен-

но; исходный текст программы может храниться в нескольких

файлах и ранее скомпилированные процедуры могут загружаться

из библиотек. Связь между функциями может осуществляться как

через явные обращения, так и в результате манипулирования с

внешними данными.

Поэтому следует рассмотреть два вида областей действия:

во-первых, ту, которая может быть названа лексической об-

ластью действия идентификатора и которая по существу являет-

ся той областью в программе, где этот идентификатор можно

использовать, не вызывая диагностического сообщения “неопре-

деленный идентификатор”; и во-вторых, область действия, ко-

торая связана с внешними идентификаторами и которая характе-

ризуется правилом, что ссылки на один и тот же внешний иден-

тификатор являются ссылками на один и тот же объект.

* 216 -

19.1. Лексическая область действия

Лексическая область действия идентификаторов, описанных

во внешних определениях, простирается от определения до кон-

ца исходного файла, в котором он находится. Лексическая об-

ласть действия идентификаторов, являющихся формальными пара-

метрами, распространяется на ту функцию, к которой они отно-

сятся. Лексическая область действия идентификаторов, описан-

ных в начале блока, простирается до конца этого блока. Лек-

сической областью действия меток является та функция, в ко-

торой они находятся.

Поскольку все обращения на один и тот же внешний иденти-

фикатор обращаются к одному и тому же объекту (см. П. 19.2),

Компилятор проверяет все описания одного и того же внешнего

идентификатора на совместимость; в действительности их об-

ласть действия распространяется на весь файл, в котором они

находятся.

Во всех случаях, однако, есть некоторый идентификатор,

явным образом описан в начале блока, включая и блок, который

образует функцию, то действие любого описания этого иденти-

фикатора вне блока приостанавливается до конца этого блока.

Напомним также (п. 16.5), Что идентификаторы, соответст-

вующие обычным переменным, с одной стороны, и идентификато-

ры, соответствующие членам и ярлыкам структур и объединений,

с другой стороны, формируют два непересекающихся класса, ко-

торые не вступают в противоречие. Члены и ярлыки подчиняются

тем же самым правилам определения областей действия, как и

другие идентификаторы. Имена, специфицируемые с помощью

TYPEDEF, входят в тот же класс, что и обычные идентификато-

ры. Они могут быть переопределены во внутренних блоках, но

во внутреннем описании тип должен быть указан явно:

TYPEDEF FLOAT DISTANCE;

...

\(

AUTO INT DISTANCE;

...

Во втором описании спецификатор типа INT должен присутство-

вать, так как в противном случае это описание будет принято

за описание без описателей с типом DISTANCE (прим. Автора:

согласитесь, что лед здесь тонок.).

19.2. Область действия внешних идентификаторов

Если функция ссылается на идентификатор, описанный как

EXTERN, то где-то среди файлов или библиотек, образующих

полную программу, должно содержаться внешнее определение

этого идентификатора. Все функции данной программы, которые

ссылаются на один и тот же внешний идентификатор, ссылаются

на один и тот же объект, так что следует позаботиться, чтобы

специфицированные в этом определении тип и размер были сов-

местимы с типом и размером, указываемыми в каждой функции,

которая ссылается на эти данные.

* 217 -

Появление ключевого слова EBTERN во внешнем определении

указывает на то, что память для описанных в нем идентифика-

торов будет выделена в другом файле. Следовательно, в состо-

ящей из многих файлов программе внешнее определение иденти-

фикатора, не содержащее спецификатора EXTERN, должно появ-

ляться ровно в одном из этих файлов. любые другие файлы, ко-

торые желают дать внешнее определение этого идентификатора,

должны включать в это определение слово EXTERN. Идентифика-

тор может быть инициализирован только в том описании, кото-

рое приводит к выделению памяти.

Идентификаторы, внешнее определение которых начинается

со слова STATIC, недоступны из других файлов. Функции могут

быть описаны как STATIC.

20. Строки управления компилятором

Компилятор языка “C” содержит препроцессор, который поз-

воляет осуществлять макроподстановки, условную компиляцию и

включение именованных файлов. Строки, начинающиеся с #, об-

щаются с этим препроцессором. Синтаксис этих строк не связан

с остальным языком; они могут появляться в любом месте и их

влияние распространяется (независимо от области действия) до

конца исходного программного файла.

20.1. Замена лексем

Управляющая компилятором строка вида

#DEFINE идентификатор строка-лексем

(Обратите внимание на отсутствие в конце точки с запя-

той) приводит к тому, что препроцессор заменяет последующие

вхождения этого идентификатора на указанную строку лексем.

Строка вида

#DEFINE идентификатор

(идентификатор,...,идентификатор)строка лексем

где между первым идентификатором и открывающейся скобкой (

нет пробела, представляет собой макроопределение с аргумен-

тами. Последующее вхождение первого идентификатора, за кото-

рым следует открывающая скобка '(', последовательность раз-

деленных запятыми лексем и закрывающая скобка ')', заменяют-

ся строкой лексем из определения. каждое вхождение идентифи-

катора, упомянутого в списке формальных параметров в опреде-

лении , заменяется соответствующей строкой лексем из обраще-

ния. Фактическими аргументами в обращении являются строки

лексем, разделенные запятыми; однако запятые, входящие в за-

кавыченные строки или заключенные в круглые скобки, не раз-

деляют аргументов. Количество формальных и фактических пара-

метров должно совпадать. Текст внутри строки или символьной

константы не подлежит замене.

В обоих случаях замененная строка просматривается снова

с целью обнаружения других определенных идентификаторов. В

обоих случаях слишком длинная строка определения может быть

продолжена на другой строке, если поместить в конце продол-

жаемой строки обратную косую черту \ .

* 218 -

Описываемая возможность особенно полезна для определения

“объявляемых констант”, как, например,

#DEFINE TABSIZE 100

INT TABLE[TABSIZE];

Управляющая строка вида

#UNDEF идентификатор

приводит к отмене препроцессорного определения данного иден-

тификатора.

20.2. Включение файлов

Строка управления компилятором вида

#INCLUDE “FILENAME”

приводит к замене этой строки на все содержимое файла с име-

нем FILENAME. Файл с этим именем сначала ищется в справочни-

ке начального исходного файла, а затем в последовательности

стандартных мест. В отличие от этого управляющая строка вида

#INCLUDE <FILENAME>

ищет файл только в стандартных местах и не просматривает

справочник исходного файла.

Строки #INCLUDE могут быть вложенными.

20.3. Условная компиляция

Строка управления компилятором вида

#IF константное выражение

проверяет, отлично ли от нуля значение константного выраже-

ния (см. П. 15). Управляющая строка вида

#IF DEF идентификатор

проверяет, определен ли этот идентификатор в настоящий мо-

мент в препроцессоре, т.е. Определен ли этот идентификатор с

помощью управляющей строки #DEFINE.

21. Неявные описания

Не всегда является необходимым специфицировать и класс

памяти и тип идентификатора в описании. Во внешних определе-

ниях и описаниях формальных параметров и членов структур

класс памяти определяется по контексту. Если в находящемся

внутри функции описании не указан тип, а только класс памя-

ти, то предполагается, что идентификатор имеет тип INT; если

не указан класс памяти, а только тип, то идентификатор пред-

полагается описанным как AUTO. Исключение из последнего пра-

вила дается для функций, потому что спецификатор AUTO для

функций является бессмысленным (язык “C” не в состоянии ком-

пилировать программу в стек); если идентификатор имеет тип

“функция, возвращающая ...”, то он предполагается неявно

описанным как EXTERN.

* 219 -

Входящий в выражение и неописанный ранее идентификатор,

за которым следует скобка ( , считается описанным по контек-

сту как “функция, возвращающая INT”.

22. Снова о типах

В этом разделе обобщаются сведения об операциях, которые

можно применять только к объектам определенных типов.

22.1. Структуры и объединения

Только две вещи можно сделать со структурой или объеди-

нением: назвать один из их членов (с помощью операции) или

извлечь их адрес ( с помощью унарной операции &). Другие

операции, такие как присваивание им или из них и передача их

в качестве параметров, приводят к сообщению об ошибке. В бу-

дущем ожидается, что эти операции, но не обязательно ка-

кие-либо другие, будут разрешены.

В п. 15.1 Говорится, что при прямой или косвенной ссылке

на структуру (с помощью . Или ->) имя справа должно быть

членом структуры, названной или указанной выражением слева.

Это ограничение не навязывается строго компилятором, чтобы

дать возможность обойти правила типов. В действительности

перед '.' допускается любое L-значение и затем предполагает-

ся, что это L-значение имеет форму структуры, для которой

стоящее справа имя является членом. Таким же образом, от вы-

ражения, стоящего перед '->', требуется только быть указате-

лем или целым. В случае указателя предполагается, что он

указывает на структуру, для которой стоящее справа имя явля-

ется членом. В случае целого оно рассматривается как абсо-

лютный адрес соответствующей структуры, заданный в единицах

машинной памяти.

Такие структуры не являются переносимыми.

22.2. Функции

Только две вещи можно сделать с функцией: вызвать ее или

извлечь ее адрес. Если имя функции входит в выражение не в

позиции имени функции, соответствующей обращению к ней, то

генерируется указатель на эту функцию. Следовательно, чтобы

передать одну функцию другой, можно написать

INT F();

...

G(F);

Тогда определение функции G могло бы выглядеть так:

G(FUNCP)

INT(\*FUNCP)();

\(

...

(\*FUNCP)();

...

\)

Обратите внимание, что в вызывающей процедуре функция F дол-

жна быть описана явно, потому что за ее появлением в G(F) не

следует скобка ( .

* 220 -

22.3. Массивы, указатели и индексация

Каждый раз, когда идентификатор, имеющий тип массива,

появляется в выражении, он преобразуется в указатель на пер-

вый член этого массива. Из-за этого преобразования массивы

не являются L-значениями. По определению операция индексация

[] интерпретируется таким образом, что E1[E2] считается

идентичным выражению \*((е1)+(е2)). Согласно правилам преоб-

разований, применяемым при операции +, если E1 - массив, а

е2 - целое, то е1[е2] ссылается на е2-й член массива е1. По-

этому несмотря на несимметричный вид операция индексации яв-

ляется коммутативной.

В случае многомерных массивов применяется последователь-

ное правило. Если е является N-мерным массивом размера

I\*J\*...\*K, то при появлении в выражении е преобразуется в

указатель на (N-1)-мерный массив размера J\*...\*K. Если опе-

рация \* либо явно, либо неявно, как результат индексации,

применяется к этому указателю, то результатом операции будет

указанный (N-1)-мерный массив, который сам немедленно преоб-

разуется в указатель.

Рассмотрим, например, описание

INT X[3][5];

Здесь X массив целых размера 3\*5. При появлении в выражении

X преобразуется в указатель на первый из трех массивов из 5

целых. В выражении X[I], которое эквивалентно \*(X+I), снача-

ла X преобразуется в указатель так, как описано выше; затем

I преобразуется к типу X, что вызывает умножение I на длину

объекта, на который указывает указатель, а именно на 5 целых

объектов. Результаты складываются, и применение косвенной

адресации дает массив (из 5 целых), который в свою очередь

преобразуется в указатель на первое из этих целых. Если в

выражение входит и другой индекс, то таже самая аргументация

применяется снова; результатом на этот раз будет целое.

Из всего этого следует, что массивы в языке “C” хранятся

построчно ( последний индекс изменяется быстрее всего) и что

первый индекс в описании помогает определить общее количест-

во памяти, требуемое для хранения массива, но не играет ни-

какой другой роли в вычислениях, связанных с индексацией.

22.4. Явные преобразования указателей

Разрешаются определенные преобразования, с использовани-

ем указателей , но они имеют некоторые зависящие от конкрет-

ной реализации аспекты. Все эти преобразования задаются с

помощью операции явного преобразования типа; см. П. 15.2 и

16.7.

Указатель может быть преобразован в любой из целочислен-

ных типов, достаточно большой для его хранения. Требуется ли

при этом INT или LONG, зависит от конкретной машины. Преоб-

разующая функция также является машинно-зависимой, но она

будет вполне естественной для тех, кто знает структуру адре-

сации в машине. Детали для некоторых конкретных машин приво-

дятся ниже.

Объект целочисленного типа может быть явным образом пре-

образован в указатель. такое преобразование всегда переводит

преобразованное из указателя целое в тот же самый указатель,

но в других случаях оно будет машинно-зависимым.

* 221 -

Указатель на один тип может быть преобразован в указа-

тель на другой тип. Если преобразуемый указатель не указыва-

ет на объекты, которые подходящим образом выравнены в памя-

ти, то результирующий указатель может при использовании вы-

зывать ошибки адресации. Гарантируется, что указатель на

объект заданного размера может быть преобразован в указатель

на объект меньшего размера и снова обратно, не претерпев при

этом изменения.

Например, процедура распределения памяти могла бы прини-

мать запрос на размер выделяемого объекта в байтах, а возв-

ращать указатель на символы; это можно было бы использовать

следующим образом.

EXTERN CHAR \*ALLOC();

DOUBLE \*DP;

DP=(DOUBLE\*) ALLOC(SIZEOF(DOUBLE));

\*DP=22.0/7.0;

Функция ALLOC должна обеспечивать (машинно-зависимым спосо-

бом), что возвращаемое ею значение будет подходящим для пре-

образования в указатель на DOUBLE; в таком случае использо-

вание этой функции будет переносимым.

Представление указателя на PDP-11 соответствует 16-бито-

вому целому и измеряется в байтах. Объекты типа CHAR не име-

ют никаких ограничений на выравнивание; все остальные объек-

ты должны иметь четные адреса.

На HONEYWELL 6000 указатель соответствует 36-битовому

целому; слову соответствует 18 левых битов и два непосредст-

венно примыкающих к ним справа бита, которые выделяют символ

в слове. Таким образом, указатели на символы измеряются в

единицах 2 в степени 16 байтов; все остальное измеряется в

единицах 2 в степени 18 машинных слов. Величины типа DOUBLE

и содержащие их агрегаты должны выравниваться по четным ад-

ресам слов (0 по модулю 2 в степени 19). Эвм IBM 370 и

INTERDATA 8/32 сходны между собой. На обеих машинах адреса

измеряются в байтах; элементарные объекты должны быть выров-

нены по границе, равной их длине, так что указатели на SHORT

должны быть кратны двум, на INT и FLOAT - четырем и на

DOUBLE - восьми. Агрегаты выравниваются по самой строгой

границе, требуемой каким-либо из их элементов.

23. Константные выражения

В нескольких местах в языке “C” требуются выражения, ко-

торые после вычисления становятся константами: после вариан-

тного префикса CASE, в качестве границ массивов и в инициа-

лизаторах. В первых двух случаях выражение может содержать

только целые константы, символьные константы и выражения

SIZEOF, возможно связанные либо бинарными операциями

+ - \* / . % & \! Ч << >> == 1= <> <= >=

либо унарными операциями

- \^

либо тернарной операцией ?:

* 222 -

Круглые скобки могут использоваться для группировки, но не

для обращения к функциям.

В случае инициализаторов допускается большая (ударение

на букву о) свобода; кроме перечисленных выше константных

выражений можно также применять унарную операцию & к внешним

или статическим объектам и к внешним или статическим масси-

вам, имеющим в качестве индексов константное выражение.

Унарная операция & может быть также применена неявно, в ре-

зультате появления неиндексированных массивов и функций. Ос-

новное правило заключается в том, что после вычисления ини-

циализатор должен становится либо константой, либо адресом

ранее описанного внешнего или статического объекта плюс или

минус константа.

24. Соображения о переносимости

Некоторые части языка “C” по своей сути машинно-зависи-

мы. Следующие ниже перечисление потенциальных трудностей хо-

тя и не являются всеобъемлющими, но выделяет основные из

них.

Как показала практика, вопросы, целиком связанные с ап-

паратным оборудованием, такие как размер слова, свойства

плавающей арифметики и целого деления, не представляют осо-

бенных затруднений. Другие аспекты аппаратных средств нахо-

дят свое отражение в различных реализациях. Некоторые из

них, в частности, знаковое расширение (преобразующее отрица-

тельный символ в отрицательное целое) и порядок, в котором

помещаются байты в слове, представляют собой неприятность,

которая должна тщательно отслеживаться. Большинство из ос-

тальных проблем этого типа не вызывает сколько-нибудь значи-

тельных затруднений.

Число переменных типа REGISTER, которое фактически может

быть помещено в регистры, меняется от машины к машине, также

как и набор допустимых для них типов. Тем не менее все ком-

пиляторы на своих машинах работают надлежащим образом; лиш-

ние или недопустимые регистровые описания игнорируются.

Некоторые трудности возникают только при использовании

сомнительной практики программирования. Писать программы,

которые зависят от каких- либо этих свойств, является чрез-

вычайно неразумным.

Языком не указывается порядок вычисления аргументов фун-

кций; они вычисляются справа налево на PDP-11 и VAX-11 и

слева направо на остальных машинах. порядок, в котором про-

исходят побочные эффекты, также не специфицируется.

Так как символьные константы в действительности являются

объектами типа INT, допускается использование символьных

констант, состоящих из нескольких символов. Однако, посколь-

ку порядок, в котором символы приписываются к слову, меняет-

ся от машины к машине, конкретная реализация оказывается

весьма машинно-зависимой.

Присваивание полей к словам и символов к целым осуществ-

ляется справо налево на PDP-11 и VAX-11 и слева направо на

других машинах. эти различия незаметны для изолированных

программ, в которых не разрешено смешивать типы (преобразуя,

например, указатель на INT в указатель на CHAR и затем про-

веряя указываемую память), но должны учитываться при согла-

совании с накладываемыми извне схемами памяти.

* 223 -

Язык, принятый на различных компиляторах, отличается

только незначительными деталями. Самое заметное отличие сос-

тоит в том, что используемый в настоящее время компилятор на

PDP-11 не инициализирует структуры, которые содержат поля

битов, и не допускает некоторые операции присваивания в оп-

ределенных контекстах, связанных с использованием значения

присваивания.

25. Анахронизмы

Так как язык “C” является развивающимся языком, в старых

программах можно встретить некоторые устаревшие конструкции.

Хотя большинство версий компилятора поддерживает такие анах-

ронизмы, они в конце концов исчезнут, оставив за собой толь-

ко проблемы переносимости.

В ранних версиях “C” для проблем присваивания использо-

валась форма =ON, а не ON=, приводя к двусмысленностям, ти-

пичным примером которых является

X = -1

где X фактически уменьшается, поскольку операции = и - при-

мыкают друг к другу, но что вполне могло рассматриваться и

как присваивание -1 к X.

Синтаксис инициализаторов изменился: раньше знак равенс-

тва, с которого начинается инициализатор, отсутствовал, так

что вместо

INT X = 1;

использовалось

INT X 1;

изменение было внесено из-за инициализации

INT F (1+2)

которая достаточно сильно напоминает определение функции,

чтобы смутить компиляторы.

26. Сводка синтаксических правил

Эта сводка синтаксиса языка “C” предназначена скорее для

облегчения понимания и не является точной формулировкой язы-

ка.

26.1. Выражения Основными выражениями являются следующие:

выражение:

первичное-выражение

* выражение

& выражение

* выражение

! Выражение

\^ выражение

++ L-значение

* L-значение

L-значение ++

L-значение—

* 224 -

SIZEOF выражение

(имя типа) выражение

выражение бинарная-операция выражение

выражение ? Выражение : выражение

L-значение операция-присваивания выражение

выражение , выражение

первичное выражение:

идентификатор

константа

строка

(выражение)

первичное-выражение (список выражений

необ)

первичное-выражение [выражение]

L-значение . Идентификатор

первичное выражение -> идентификатор

L-значение:

идентификатор

первичное-выражение [выражение]

L-значение . Идентификатор

первичное-выражение -> идентификатор

* выражение

(L-значение)

Операции первичных выражений

() [] . ->

имеют самый высокий приоритет и группируются слева

направо. Унарные операции

* & - ! \^ ++ -- SIZEOF(Имя типа)

имеют более низкий приоритет, чем операции первичных выраже-

ний, но более высокий, чем приоритет любой бинарной опера-

ции. Эти операции группируются справа налево. Все бинарные

операции и условная операция (прим. Перевод.: условная опе-

рация группируется справа налево; это изменение внесено в

язык в 1978 г.) группируются слева направо и их приоритет

убывает в следующем порядке:

Бинарные операции:

\* / %

+ -

>> <<

< > <= >=

== !=

&

\^

\!

&&

\!\!

?:

* 225 -

Все операции присваивания имеют одинаковый приоритет и груп-

пируются справа налево.

Операции присваивания:

= += -= \*= ?= %= >>= <<= &= \^= \!=

Операция запятая имеет самый низкий приоритет и группируется

слева направо.

26.2. Описания Описание:

спецификаторы-описания список-инициализируемых-описателей

необ;

спецификаторы-описания:

спецификатор-типа спецификаторы-описания

необ

спецификатор-класса-памяти спецификаторы-описания

необ

спецификатор-класса-памяти:

AUTO

STATIC

EXTERN

REGISTER

TYPEDEF

спецификатор-типа:

CHAR

SHORT

INT

LONG

UNSIGNED

FLOAT

DOUBLE

спецификатор-структуры-или-объединения

определяющее-тип-имя

список-инициализируемых-описателей:

инициализируемый-описатель

инициализируемый-описатель,

список-инициализируемых-описателей

инициализируемый-описатель

описатель-инициализатор

необ

описатель:

идентификатор

(описатель)

* описатель описатель () описатель [константное выражение

необ]

* 226 -

спецификатор-структуры-или-объединения:

STRUCT список-описателей-структуры

STRUCT идентификатор \(список-описаний-структуры\)

STRUCT идентификатор

UNION \(список-описаний-структуры\)

UNION идентификатор \(список-описаний-структуры\)

UNION идентификатор

список-описаний-структцры:

описание-структуры

описание-структуры список-описаний-структуры

описание структуры:

спецификатор-типа список-описателей-структуры:

список-описателей-структуры

описатель-структуры

описатель-структуры,список-описателей-структуры

описатель-структуры:

описатель

описатель: константное выражение

:константное-выражение

инициализатор:

= выражение

= \(список-инициализатора\)

= \(список-инициализатора\)

список инициализатора:

выражение

список-инициализатора,список-инициализатора

\(список-инициализатора\)

имя-типа:

спецификатор-типа абстрактный-описатель

абстрактный-описатель:

пусто

\(абстрактный-описатель\)

* абстрактный-описатель абстрактный-описатель () абстрактный-описатель [константное-выражение

необ]

определяющее-тип-имя:

идентификатор

26.3. Операторы

составной-оператор:

\(список-описаний список-операторов

необ необ\)

список-описаний:

описание

описание список-описаний

список-операторов:

оператор

оператор список-операторов

оператор:

составной оператор

выражение;

* 227 -

IF (выражение) оператор

IF (выражение) оператор ELSE оператор

WHILE (выражение) оператор

DO оператор WHILE (выражение);

FOR(выражение-1 ;выражение-2 ;выражение-3 )

необ необ необ

оператор

SWITCH (выражение) оператор

CASE константное-выражение : оператор

DEFAULT: оператор

BREAK;

CONTINUE;

RETURN;

RETURN выражение;

GOTO идентификатор;

идентификатор : оператор

;

26.4. Внешние определения Программа:

внешнее-определение

внешнее-определение программа

внешнее-определение:

определение-функции

определение-данных

определение-функции:

спецификатор-типа описатель-функции тело-функции

необ

описатель-функции:

описатель (список-параметров )

необ

список-параметров:

идетификатор

идентификатор , список-параметров

тело-функции:

список-описаний-типа оператор-функции

оператор-функции:

\(список описаний список-операторов\)

необ

определение данных:

EXTERN спецификатор типа список

необ необ

инициализируемых описателей ;

необ

STATIC спецификатор типа список

необ необ

инициализируемых описателей

необ;

26.5. Препроцессор #DEFINE идентификатор строка-лексем #DEFINE

* 229 -

#DEFINE идентификатор(идентификатор,...,идентификатор)стр

#UNDEF идентификатор

#INCLUDE “имя-файла”

#INCLUDE <имя-файла>

#IF константное-выражение

#IFDEF идентификатор

#IFNDEF идентификатор

#ELSE

#ENDIF

#LINE константа идентификатор

Последние изменения языка “C” (15 ноября 1978 г.)

27. Присваивание структуры

Структуры могут быть присвоены, переданы функциям в ка-

честве аргументов и возвращены функциям. Типы участвующих

операндов должны оставаться теми же самыми. Другие правдопо-

добные операторы, такие как сравнение на равенство, не были

реализованы.

В реализации возвращения структур функциями на PDP-11

имеется коварный дефект: если во время возврата происходит

прерывание и та же самая функция пеентерабельно вызывается

во время этого прерывания, то значение возвращаемое из пер-

вого вызова, может быть испорчено. Эта трудность может воз-

никнуть только при наличии истинного прерывания, как из опе-

рационной системы, так и из программы пользователя, прерыва-

ния, которое существенно для использования сигналов; обычные

рекурсивные вызовы совершенно безопасны.

28. Тип перечисления

Введен новый тип данных,аналогичный скалярным типам язы-

ка паскаль. К спецификатору-типа в его синтаксическом описа-

нии в разделе 8.2. Приложения а следует добавить

спецификатор-перечисления

с синтаксисом

пецификатор-перечисления:

ENUM список-перечисления

ENUM идентификатор список-перечисления

------------- -------------------

ENUM идентификатор

* 230 -

cписок-перечисления:

перечисляемое

список-перечисления, перечисляемое

------------------- -------------

перечисляемое:

идентификатор

идентификатор = константное выражение

------------- ---------------------

Роль идентификатора в спецификаторе-перечисления пол-

ностью аналогична роли ярлыка структуры в спецификато-

ре-структуры; идентификатор обозначает определенное перечис-

ление. Например, описание

ENUM COLOR \(RED, WHITE, BLACK, BLUE \);

. . .

ENUM COLOR \*CP, COL;

Объявляет идентификатор COLOR ярлыком перечисления типа,

описывающего различные цвета и затем объявляет CP указателем

на объект этого типа, а COL - объектом этого типа.

Идентификаторы в списке-перечисления описываются как

константы и могут появиться там, где требуются (по контекс-

ту) константы. Если не используется вторая форма перечисляе-

мого (с равеством =), то величины констант начинаются с 0 и

возрастают на 1 в соответствии с прочтением их описания сле-

ва на право. Перечисляемое с присвоением = придает соответс-

твующему идентификатору указанную величину; последующие

идентификаторы продолжают прогрессию от приписанной величи-

ны.

Все ярлыки перечисления и константы могут быть различны-

ми и непохожими на ярлыки и члены структур даже при условии

использования одного и того же множества идентификаторов.

Объекты данного типа перечисления рассматриваются как

объекты, имеющие тип, отличный от любых типов и контролирую-

щая программа LINT сообщает об ошибках несоответствия типов.

В реализации на PDP-11 со всеми перечисляемыми переменными

оперируют так, как если бы они имели тип INT.

29. Таблица изображений непечатных символов языка “C”.

В данной таблице приведены изображения некоторых симво-

лов (фигурные скобки и т.д.) языка “C”, которых может не

оказаться в знаковом наборе дисплея или печатающего устройс-

тва.

* 231 -

! Значение ! Изображение \*\* !

! ! В тексте !

! Фигурная открывающаяся ! !

! Скобка ! \( !

! ! !

! Фигурная закрывающаяся ! !

! Скобка ! \) !

! ! !

! Вертикальная ! !

! Черта ! \! !

! ! !

! ! !

! Апостороф ! \' !

! ! !

! Волнистая ! !

! Черта ! \^ !

! ! !

\*\* П\_р\_и\_м\_е\_ч\_а\_н\_и\_е:

Изображения приведены для операционой системы UNIX. При

работе компилятора “C” под управлением любой другой операци-

онной системы, необходимо воспользоваться соответствующим

руководством для данной системы.