[17. Библиотека языка С и ввод-вывод.](#_Toc515770369)

[17.1. Стандартные библиотеки С и C++.](#_Toc515770370)

[17.2. Доступ в библиотеку языка Си.](#_Toc515770371)

[17.2.1. Автоматический доступ.](#_Toc515770372)

[17.2.2. Включение файла.](#_Toc515770373)

[17.2.3. Включение библиотеки.](#_Toc515770374)

[17.3. Потоковый ввод-вывод.](#_Toc515770375)

[17.4. Связь с файлами.](#_Toc515770376)

[17.5. Понятие файла.](#_Toc515770377)

[17.5.1. Текстовые и бинарные (двоичные) файлы](#_Toc515770378)

[17.6. Потоковые функции.](#_Toc515770379)

[17.6.1. Потоковый ввод-вывод. Стандартные потоки](#_Toc515770380)

[17.6.2. Открытие потоков.](#_Toc515770381)

[17.6.3. Файлы.](#_Toc515770382)

[17.6.4. Текстовые файлы с буферизацией.](#_Toc515770383)

[17.6.5. Переназначение ввода и вывода.](#_Toc515770384)

[17.6.6. Изменение буфера потока.](#_Toc515770385)

[17.6.7. Закрытие потоков.](#_Toc515770386)

[17.6.8. Функции, осуществляющие ввод-вывод в файловый поток](#_Toc515770387)

[17.7. Низкоуровневый ввод и вывод в С.](#_Toc515770388)

[17.8. Ввод и вывод символов.](#_Toc515770389)

[17.8.1. Использование функций getc(), putc(), fgetc() и fputc().](#_Toc515770390)

[17.8.1.1. Простая программа сжатия файла.](#_Toc515770391)

[17.8.2. Использование функций getchar(), putchar(),fgetchar() и fputchar().](#_Toc515770392)

[17.8.3. Использование функций getch() и putch().](#_Toc515770393)

[17.9. Определение строк в программе.](#_Toc515770394)

[17.9.1. Строковые константы.](#_Toc515770395)

[17.9.2. Массивы символьных строк и их инициализация.](#_Toc515770396)

[17.9.3. Различия: массив и указатель.](#_Toc515770397)

[17.9.4. Явное задание размера памяти.](#_Toc515770398)

[17.9.5. Массивы символьных строк.](#_Toc515770399)

[17.9.6. Указатели и строки.](#_Toc515770400)

[17.10. Ввод и вывод строк.](#_Toc515770401)

[17.10.1. Выделение памяти.](#_Toc515770402)

[17.10.2. Использование функций gets(), puts(), fgets() и fputs().](#_Toc515770403)

[17.11. Функции, работающие со строками.](#_Toc515770404)

[17.11.1. Функция strlen().](#_Toc515770405)

[17.11.2. Функция strcat().](#_Toc515770406)

[17.11.3. Функция strcmp().](#_Toc515770407)

[17.11.4. Функция strcpy().](#_Toc515770408)

[17.12. Создание собственных функций ввода/вывода.](#_Toc515770409)

[17.13. Проверка и преобразование символов.](#_Toc515770410)

[17.14. Преобразования символьных строк: atoi(), atof().](#_Toc515770411)

[17.15. Ввод и вывод целых чисел.](#_Toc515770412)

[17.15.1. Использование функций getw() и putw().](#_Toc515770413)

[17.16. Форматированный вывод.](#_Toc515770414)

[17.16.1. Использование функций printf() и fprintf().](#_Toc515770415)

[17.17. Использование функций fread и fwrite, fseek(), ftell() и rewind().](#_Toc515770416)

[17.17.1. Использование встроенного отладчика.](#_Toc515770417)

[17.18. Форматированный ввод.](#_Toc515770418)

[17.19. Использование функций scanf(), fscanf() и sscanf().](#_Toc515770419)

[17.20. Распределение памяти: malloc() и calloc().](#_Toc515770420)

[17.21. Другие библиотечные функции.](#_Toc515770421)

1. Библиотека языка С и ввод-вывод.

Всякий раз, когда нам нужно использовать такие функции, как printf(), getchar() и strlen(), мы обращаемся в библиотеку языка Си. Она содержит множество функций и макроопределений. Библиотеки меняются от системы к системе, но есть ядро функций (называемое стандартной библиотекой), которое используется чаше всего. Здесь мы рассмотрим пятнадцать наиболее общих из этих функций, уделяя больше внимания функциям ввода-вывода и использованию файлов.

Однако сначала давайте поговорим о том, как пользоваться библиотекой.

## Стандартные библиотеки С и C++.

Некоторые действия постоянно выполняются во многих программах и программируются практически всеми разработчиками. Для примера можно взять операцию извлечения квадратного корня. В математических функциях для извлечения квадратного корня используются комбинации основных арифметических операций: сложения, вычитания, умножения и деления.

Каждому программисту бессмысленно создавать собственную процедуру для вычисления корня и затем встраивать ее в программу. Подобные проблемы решены в С и C++ при помощи библиотек функций для выполнения подобных вычислений. Имея библиотеку, достаточно написать один оператор с вызовом нужной функции.

В этом разделе рассматриваются функции, обычно поставляемые с компилятором С и C++. Чаще всего эти функции существуют не в виде исходного текста, а в скомпилированном виде. При выполнении компоновки для получения законченной программы код библиотечных функций объединяется с откомпилированным кодом, написанным программистом.

Библиотечные функции выполняют не только математические действия, но и другие, часто встречающиеся операции. Например, имеются библиотечные функции для чтения и записи файлов на дисках, управления памятью, ввода/вывода и множества других операций. Библиотеки не являются частью стандартного С или C++, но практически каждая система проектирования предлагает определенные библиотечные функции.

Большинство библиотечных функций написаны так, что они используют информацию, содержащуюся в определенных файлах, поставляемых с системой. Следовательно, при использовании библиотек эти файлы должны быть включены в программу и обрабатываться компилятором Visual C/C++. Обычно подобные файлы имеют расширение .h и называются заголовочными файлами. В табл. 6.7 перечислены заголовочные файлы, поставляемые с Microsoft Visual C/C++.

Таблица 6.7. Заголовочные файлы Microsoft Visual C/C++

|  |  |
| --- | --- |
| **Название заголовочного файла** | **Описание** |
| assert.h | Макрос отладчика assert |
| bios.h | Сервисные функции BIOS |
| cderr.h | Коды ошибок окон диалога |
| colordlg.h | Идентификационные (ID) номера цветов управляющих элементов окон диалога |
| commdlg.h | Функции, типы и описания окон диалога |
| conio.h | Функции консоли и портов ввода/вывода |
| cpl.h | Описание панели управления и DLL |
| ctype.h | Классификация символов |
| custcntl.h | Библиотека пользовательских элементов управления |
| dde.h | Динамический обмен данными |
| ddeml.h | Интерфейс прикладного программирования DDEML |
| direct.h | Управление каталогами |
| dlgs.h | Идентификационные номера элементов окон диалога |
| dos.h | Интерфейс MS-DOS |
| drivinit.h | Устаревший — используйте print.h |
| errno.h | Описания переменной errno (код ошибки) |
| excpt.h | Структурированная обработка исключительных ситуаций |
| fcntl.h | Флаги, используемые в \_open и \_sopen |
| float.h | Константы, используемые автоматическими функциями |
| fpieee.h | Обработка исключительных ситуаций для чисел с плавающей точкой в стандарте IEEE |
| fstream.h | Функции для классов filebuf и fstream |
| graph.h | Низкоуровневая графика и шрифты |
| io.h | Управление файлами и низкоуровневый ввод/вывод |
| iomanip.h | Параметрические манипуляторы для iostream |
| ios.h | Функции для класса ios |
| iostream.h | Функции для классов iostream |
| istream.h | Функции для класса istream |
| limits.h | Диапазоны целых чисел и символов |
| locale.h | Функции локализации |
| lzdos.h | Устаревший — заменен при помощи #define lib на #include<lzexpand.h> |
| lzexpand.h | Интерфейс Public для lzexpand.dll |
| malloc.h | Функции выделения памяти |
| math.h | Математические функции для чисел с плавающей точкой |
| memory.h | Функции работы с буферами |
| mmsystem.h | Интерфейс прикладного программирования для мультимедиа |
| new.h | Функции выделения памяти С++ |
| ntimage.h | Структуры изображений |
| ntsdexts.h | Расширения отладчика NTSD и KD |
| ole.h | Функции, типы и описания OLE |
| ostream.h | Функции для класса ostream |
| penwin.h | Функции, типы и описания для оконных перьев |
| penwoem.h | Интерфейсы API для распознавания оконных перьев |
| pgchart.h | Презентационная графика |
| print.h | Функции печати, типы и описания |
| process.h | Управление процессами |
| rpc.h | Приложения RPC (Remote Procedure Call — вызов удаленных процедур) |
| rpcdce.h | Интерфейсы API для выполнения RPC в среде распределенных вычислений (DCE) |
| rpcdcep.h | Интерфейсы API для выполнения частных (private) RPC |
| rpcndr.h | Преобразование чисел с плавающей точкой и чисел двойной длины для RPC |
| rpcnsi.h | Данные интерфейса API, независимого от службы имен |
| rpcnsip.h | Описания типов и функций для средств автоматической настройки при выполнении RPC |
| rpcnterr.h | Коды ошибок компилятора и выполнения RPC |
| scmsave.h | Директивы define и описания хранителей экрана Windows 3.1 |
| search.h | Функции поиска и сортировки |
| setjmp.h | Функции setjmp и longjmp |
| share.h | Флаги, используемые в \_sopen |
| shellapi.h | Функции, типы и описания для shell.dll |
| signal.h | Константы, используемые функцией signal |
| stdarg.h | Макросы для функций со списком аргументов переменной длины |
| stddef.h | Общие типы данных и значения |
| stdio.h | Стандартный ввод/вывод |
| stdiostr.h | Функции для классов stdiostream и stdiobuf |
| stdlib.h | Общие библиотечные функции |
| streamb.h | Функции для класса streambuf |
| stress.h | Функции ударения |
| string.h | Обработка строк |
| strstrea.h | Функции для классов strstream и strstreambuf |
| tchar.h | Общие интернациональные функции |
| time.h | Функции времени |
| toolhelp.h | Функции, типы и описания для toolhelp.dll |
| varargs.h | Функции списка аргументов переменной длины |
| ver.h | Функции, типы и описания для управлениями версиями |
| vmemory.h | Виртуальная память |
| wchar.h | Широкие символы |
| wfext.h | Расширения для Диспетчера Файлов Windows |
| winbase.h | Базовые 32-разрядные Windows интерфейсы API |
| wincon.h | Подсистема консоли NT |
| windef.h | Базовые типы окон |
| windows.h | Функции, типы и описания Windows |
| windowsx.h | Макросы интерфейсов API |
| winerror.h | Коды ошибок для интерфейсов API win32 |
| wingdi.h | Данные компоненты интерфейса графического устройства GDI |
| winioctl.h | Коды управления вводом/выводом для 32-разрядных устройств Windows |
| winmem32.h | Прототипы и общие определения для winmem32.dll |
| winmm.h | Приложения мультимедиа |
| winnetwk.h | Стандартный заголовочный файл winnet для nt-win32 |
| winnls.h | Процедуры, константы и макросы для компоненты NLS (National Language Support — поддержка национальных языков) |
| winnt.h | 32-разрядные типы и константы Windows |
| winperf.h | Данные для утилиты Performance Monitor |
| winreg.h | Данные интерфейса API 32-разрядных регистров Windows |
| winsock.h | Используется для winsock.dll |
| winspool.h | API-интерфейсы печати |
| winsvc.h | Диспетчер служб |
| winuser.h | Процедуры, константы и макросы для пользовательских компонентов |
| winver.h | Для использования с ver.dll |
| sys/locking.h | Установка флагов функций |
| sys/stat.h | Статус файлов |
| sys/timeb.h | Функция времени |
| sys/types.h | Типы статусов файла и времени |
| sys/utime.h | Функция времени выполнения |

В общем, различные библиотечные функции требуют разных заголовочных файлов. Заголовочные файлы, необходимые для некоторой функции, перечислены в ее описании. Например: для функции sqrt() необходимы объявления, находящиеся в заголовочном файле math.h. Все библиотечные функции и связанные с ними заголовочные файлы описаны в справочнике Microsoft Visual C/C++ Run-Time Library Reference.

Далее приведен краткий список библиотек компилятора Visual C/C++, объединенных в функциональные группы:

* Функции классифицирующие
* Функции преобразования типов
* Функции управления каталогами
* Функции диагностики
* Функции графики
* Функции ввода/вывода
* Функции интерфейсов (DOS, 8086, BIOS)
* Функции манипулирования с данными
* Функции математические
* Функции выделения памяти
* Функции управления процессами
* Функции стандартные
* Функции отображения текстовых окон
* Функции времени и дат

Посмотрите в вашем справочном руководстве детальное описание отдельных функций, имеющихся в каждой библиотеке. После чтения этой главы вы должны разбираться в основных типах данных и операциях языка С.

## Доступ в библиотеку языка Си.

Получение доступа к библиотеке зависит от системы, поэтому вам нужно посмотреть в своей системе, как применять наиболее распространенные операторы. Во-первых, есть несколько различных мест расположения библиотечных функций. Например, getchar() обычно задают как макроопределение в файле stdio.h, в то время как strlen() обычно хранится в библиотечном файле. Во-вторых, различные системы имеют разные способы доступа к этим функциям. Вот три из них.

### Автоматический доступ.

Во многих больших системах UNIX вы только компилируете программы, а доступ к более общим библиотечным функциям выполняется автоматически.

### Включение файла.

Если функция задана как макроопределение, то можно директивой #include включить файл, содержащий ее определение. Часто подобные функции могут быть собраны в соответствующим образом названный заголовочный файл. Например, некоторые системы имеют файл ctype.h, содержащий макроопределения, задающие тип символа: прописная буква, цифра и т. д.

### Включение библиотеки.

На некотором этапе компиляции или загрузки программы вы можете выбрать библиотеку. В нашей системе, например, есть файл lc.lib, содержащий скомпилированную версию библиотечных функций, и мы предлагаем редактору связей IBM PC использовать эту библиотеку. Даже система, которая автоматически контролирует свою стандартную библиотеку, может иметь другие библиотеки редко применяемых функций, и эти библиотеки следует запрашивать явно, указывая соответствующий признак во время компиляции.

Очевидно, мы не сможем рассмотреть все особенности всех систем, но эти три примера должны показать, что вас ожидает. Теперь мы готовы к рассмотрению некоторых функций.

Добавим функции открытия и закрытия файлов, связи с файлами, проверки и преобразования символов, преобразования строк, функцию выхода и функции распределения памяти.

## Потоковый ввод-вывод.

Многие широко используемые языки высокого уровня имеют механизмы ввода/вывода, позволяющие создавать нетривиальные алгоритмы для получения и отображения сложных структур данных. Это не относится к языку С, в котором имеется весьма развитая библиотека функций ввода/вывода, хотя, исторически, ввод/вывод никогда не являлся частью самого языка С. Если вы пользуетесь только простыми операторами ввода/вывода, подобными операторам Паскаля readln и writeln, то такая ситуация может вас удивить. Здесь описываются более 20 различных способов организации ввода/вывода в С.

Библиотека стандартных функций ввода/вывода С позволяет считывать данные и записывать их в файлы и устройства. Однако в самом языке С отсутствуют какие-либо предопределенные файловые структуры. В С все данные обрабатываются как последовательность байт. Имеется три основных типа функций ввода/вывода: потоковые, работающие с консолью и портами, низкоуровневые.

В потоковых функциях ввода/вывода файлы или объекты данных рассматриваются как поток отдельных символов. Выбирая соответствующую потоковую функцию, вы можете обрабатывать данные любого необходимого размера или формата, начиная от отдельных символов и заканчивая большими, сложными структурами данных.

На техническом уровне, когда программа открывает файл для ввода/вывода при помощи потоковых функций, открытый файл связывается с некоторой структурой типа FILE (предопределенной в stdio.h), содержащей базовую информацию об этом файле. После открытия потока возвращается указатель на файловую структуру. Указатель файла, иногда называемый указателем потока или потоком, используется для ссылки к файлу при всех последующих операциях ввода/вывода.

Все потоковые функции ввода/вывода обеспечивают буферизированный, форматированный или неформатированный ввод и вывод. Буферизированный поток обеспечивает место для промежуточного хранения всей информации, вводимой из потока или записываемой в поток.

Поскольку дисковый ввод/вывод занимает довольно много времени, буферизация потока разгружает приложение. Вместо того чтобы вводить данные из потока по одному символу или по одному элементу данных, потоковые функции ввода/вывода получают данные поблочно. Когда приложению необходимо обработать введенную информацию, оно просто обращается к буферу, что гораздо быстрее. Когда буфер становится пустым, выполняется считывание с диска другого блока.

Во многих языках высокого уровня существует одна проблема с буферизированным вводом/выводом, которую нужно принимать во внимание. Например: если ваша программа выполнила несколько операторов вывода, которые не заполнили буфер вывода, и запись на диск не произошла, то по завершении программы эта информация будет потеряна. Для решения этой проблемы обычно выполняется вызов соответствующей функции для очистки буфера. В отличие от других языков высокого уровня, в языке С данная проблема с буферизированным вводом/выводом решается путем автоматической очистки содержимого буфера по завершении программы. Конечно, хорошо написанное приложение не должно рассчитывать на эти автоматические действия; все действия программы должны описываться в явном виде. Дополнительное замечание: если вы используете потоковый ввод/вывод и приложение заканчивается с аварийным остановом, то буферы вывода могут оказаться неочищенными, что приведет к потере данных.

Аналогичным образом выглядят процедуры, работающие с консолью и портами; их можно рассматривать как расширенные потоковые функции. Они позволяют читать и писать на терминал (консоль) или в порт ввода/вывода (например, в порт принтера). Функции портов ввода/вывода выполняют простое побайтное считывание и запись. Функции ввода/вывода на консоль обеспечивают несколько дополнительных возможностей. Например, можно определить: введен ли с консоли символ или имеют ли вводимые символы эхо-отображение на экране. Последним типом ввода и вывода является низкоуровневый. Функции низкоуровнего ввода/вывода не выполняют никакой буферизации и форматирования; они непосредственно обращаются к средствам ввода и вывода операционной системы. Эти функции позволяют обращаться к файлам и периферийным устройствам на более низком уровне, чем это делают потоковые функции. При открытии файла на этом уровне возвращается описатель файла (file handle), представляющий собой целое число, использующееся затем для обращения к этому файлу при последующих операциях. В общем случае не рекомендуется смешивать функции потокового ввода/вывода с низкоуровневыми. Поскольку потоковые функции являются буферизированными, а низкоуровневые — нет, при обращении к файлу или устройству при помощи двух разных способов возможны рассогласование или даже потеря данных в буферах. Поэтому для каждого конкретного файла необходимо использовать либо потоковые, либо низкоуровневые функции. В табл. 17.1 перечислены наиболее часто используемые в С функции потокового ввода/вывода.

Таблица 17.1. Функции С потокового ввода и вывода

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| clearer() | Сбрасывает индикатор ошибок потока и устанавливает в ноль индикатор конца файла (end-of-file) |
| fclose() | Закрывает поток |
| fcloseall() | Закрывает все потоки |
| fdopen() | Открывает поток, используя его описатель (handle), полученный при помощи creat, dup, dup2 или open |
| feof() | Проверяет поток на признак конца файла |
| ferror() | Проверяет поток на признак конца файла |
| fflush() | Проверяет поток на признак конца файла |
| fgetc() | Считывает символ из потока |
| fgetchar() | Считывает символ из потока stdin |
| fgetpos() | Возвращает текущий указатель файла |
| fgets() | Возвращает строку из потока |
| filelength() | Возвращает размер потока в байтах |
| fileno() | Возвращает описатель файла, связанного с потоком |
| flushall() | Очищает буферы всех потоков |
| fopen() | Открывает поток |
| fprintf() | Выполняет форматированную запись в поток |
| fputc() | Записывает символ в поток |
| fputchar() | Записывает символ в stdout |
| fputs() | Записывает строку в поток |
| fread() | Считывает неформатированные данные из потока |
| freopen() | Переназначает указатель потока |
| fscanf() | Считывает форматированные данные из потока |
| fseek() | Позиционирует указатель файла в указанную позицию |
| fsetpos() | Позиционирует указатель файла потока |
| fstat() | Возвращает информацию об открытом файле |
| ftell() | Возвращает положение текущего указателя файла |
| fwrite() | Записывает неформатированные данные в поток |
| getc() | Этот макрос считывает символ из потока |
| getchar() | Этот макрос считывает символ из stdin |
| gets() | Возвращает строку из stdin |
| getw () | Считывает целое число из потока |
| perror() | Выдает системную ошибку на stderr |
| printf() | Записывает форматированные данные в stdout |
| putc() | Этот макрос записывает символ в поток |
| putchar() | Этот макрос записывает символ в stdout |
| puts() | Записывает строку в stdout |
| putw() | Записывает целое число в поток |
| remove() | Удаляет файл |
| rename() | Переименует файл |
| rewind() | Позиционирует указатель файла на начало потока |
| scanf() | Сканирует и вводит форматированные данные из stdln |
| setbuf() | Отменяет автоматическую буферизацию, позволяя приложению определить собственный буфер для потока |
| setvbuf() | То же самое, что и setbuf(); позволяет также определить размер буфера |
| sprintf() | Записывает форматированные данные в строку |
| sscanf() | Сканирует и вводит форматированные данные из строки |
| tmpnam() | Создает в заданном подкаталоге файл с уникальным именем |
| ungetch() | Засылает символ в буфер клавиатуры |
| vfprintf() | Записывает форматированные данные в поток, используя указатель на форматирующую строку |
| vfscanf() | Сканирует и форматирует ввод из потока, используя указатель на форматирующую строку |
| vprintf() | Записывает форматированные данные в stdout, используя указатель на форматирующую строку |
| vscanf() | Сканирует и форматирует ввод из stdin, используя указатель на форматирующую строку |
| vsprintf() | Записывает форматированные данные в строку, используя указатель на форматирующую строку |
| vsscanf() | Сканирует и форматирует ввод строки, используя указатель на форматирующую строку |

## Связь с файлами.

Часто нам бывает нужна программа получения информации от файла или размещения результатов в файле. Один способ организации связи программы с файлом заключается в использовании операций переключения < и >. Этот метод прост, но ограничен. Например, предположим, вы хотите написать диалоговую программу, которая спрашивает у вас названия книг (звучит фамильярно?), и вы намерены сохранить весь список в файле. Если вы используете переключение как, например, в

books > bklist

то ваши диалоговые приглашения также будут переключены на bklist. И тогда не только нежелательная чепуха запишется в bklist, но и пользователь будет избавлен от вопросов, на которые он, как предполагалось, должен отвечать.

К счастью, язык Си предоставляет и более мощные методы связи с файлами. Один подход заключается в использовании функции fopen(), которая открывает файл, затем применяются специальные функции ввода-вывода для чтения файла или записи в этот файл и далее используется функция fclose() для закрытия файла. Однако прежде чем исследовать эти функции, нам нужно хотя бы кратко познакомиться с сущностью файла.

## Понятие файла.

Интуитивное определение файла звучит примерно так. **Файл** -- именованная область на жестком диске. На самом деле с точки зрения ОС UNIX это совсем не так. В ОС UNIX файл -- очень удобная абстракция. С точки зрения UNIX файлом называется "что-нибудь", из чего можно считывать информацию или во что можно записывать информацию. Файлы это:

* Файлы в обычном смысле: файлы, которые хранятся на жестком диске (можно считывать из них и запиcывать в них информацию);
* Экран монитора: файл, в который можно выводить информацию (отобразится на экране монитора);
* Клавиатура: файл, из которого можно считывать информацию;
* Принтер: файл, в который можно выводить информацию (печать текста);
* Модем: файл, из которого можно считывать информацию и в который можно записывать информацию (обмен информации по сети);

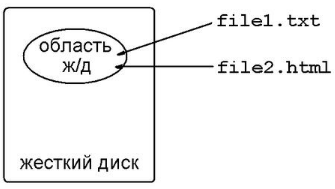
**Файл** -- это всё, что предназначено для ввода или вывода информации.

С этой точки зрения файлы бывают разными: принтер может только выводить информацию, а клавиатура -- только вводить. У такого рода файлов есть много особенностей. У файла на жестком диске есть понятие конца файла. Мы можем его считывать до тех пор, пока он не кончится. Тогда как у клавиатуры нет конца.

Неправильно думать, что между сущностями "файл" и "название файла" есть взаимно однозначное соответствие.

Можно привести аналогию из жизни: если представить, что файл -- это банка с некоторым содержимым, то название файла -- это этикетка на этой банке. Логично предположить, что у банки может быть несколько этикеток.

С точки зрения UNIX:



Правильно говорить, что у названия есть файл. И наоборот: неправильно говорить, что у файла есть название. Никакого эффективного способа узнать имя файла не существует (но можно перебрать все файлы файловой системы).

**Чтение и запись: printf и scanf**

Всем хорошо известная функция printf:

printf("Hello!") -- печать текста на экран;

printf("N = %d", N) -- форматированный вывод на экран: вывести число N в десятичной записи;

printf("N = %x", N) -- форматированный вывод на экран: вывести число N в шестнадцатеричной записи;

Аналогично парная функция scanf:

scanf("%d", &N) -- считывание с клавиатуры значения переменной N в десятичной записи;

char \*ptr = new char[10];

scanf("%s", ptr); -- считывание с клавиатуры строки в массив \*ptr

Тут могут возникать различные проблемы.

1. **Проблема безопасности:**

char \*ptr = new char[10];

scanf("%s", ptr);

Тут налицо потенциальная проблема переполнения буфера (в данном примере в буфере всего 10 байт).

Никогда не следует пользоваться scanf-ом для чтения строк.

**scanf + "%s" -- запрещенная комбинация!**

1. **Форматная строка не компилируется:** она будет разбираться в момент исполнения программы. Это обозначает проблему быстродействия. scanf -- не предназначен для чтения большого количества информации. Аналогично printf -- тоже сравнительно медленный (однако существенно быстрее, чем scanf).
2. **Проблема безопасной работы со стеком**:

printf("%d %d", N);

Проблема состоит в том, что форматная строка "%d %d" будет проанализирована в момент исполнения. В данном случае произойдет ошибка при работе со стеком: во время исполнения будет взят лишний int.

Перечисленные недостатки означают, что использование функций printf и scanf небезопасно и малоэффективно. Существенным плюсом этих функций является возможность простого форматированного ввода и вывода.

Для нас файл является частью памяти, обычно на диске, со своим именем. Мы считаем, например, stdio.h именем файла, содержащего некоторую полезную информацию. Для операционной системы файл более сложен, но это системные проблемы, а не наши. Однако мы должны знать, что означает файл для программы на языке Си. В предлагаемых для обсуждения функциях, работающих с файлами, язык Си «рассматривает» файл как структуру. Действительно, файл stdio.h содержит определение структуры файла. Вот типичный пример, взятый из IBM-версии компилятора Lattice С:

struct \_iobuf

{

char \*\_ptr; /\* текущий указатель буфера \*/

int \_cnt; /\* текущий счетчик байтов \*/

char \*\_ base; /\* базовый адрес буфера ввода-вывода\*/

char \_ flag; /\* управляющий признак \*/

char \_ file; /\* номер файла \*/

};

#define FILE struct \_iobuf /\* краткая запись \*/

Здесь мы не собираемся разбираться детально в этом определении. Главное состоит в том, что файл является структурой, и что краткое наименование шаблона файла — FILE. (Многие системы используют директиву typedef для установления этого соответствия.) Таким образом, программа, имеющая дело с файлами, будет использовать тип структуры FILE, чтобы делать так.

Имея это в виду, мы сможем лучше понять операции над файлами.

**Чтение и запись файлов: FILE\*, fopen, fprintf, fscanf**

Есть несколько способов работы с файлами c использованием языков C и C++.

Самый распространенный связан со структурой FILE (это не класс, потому что сущность языка C). Эта структура определена в заголовочном файле стандартной библиотеки <stdio.h>. Размер этой структуры и ее поля зависят от ОС и от версии компилятора. Поэтому никто не пользуется структурой FILE. Обычно пользуются указателем на эту структуру: FILE\*. Например:

FILE \*f = fopen("file1.txt", "r");

fopen -- функция из стандартной библиотеки. Первый параметр -- имя файла (в текущем каталоге). Второй параметр задает режим открытия файла; в данном случае "r" означает, что файл будет открыт только для чтения. Эта функция возвращает ненулевой указатель, если открытие прошло успешно; и возвращает NULL, если произошла ошибка. Ошибка может возникать в следующих ситуациях:

* не существует файла;
* у программы недостаточно прав доступа для работы с файлом;

Для дальнейшей корректной работы следует писать примерно такой код:

if (f == NULL) {

// файл не удалось открыть

}

else {

// Работа с файлом

}

Допустим, что нам удалось открыть файл, т.е. f != NULL. Тогда для того, чтобы считывать файл, можно использовать функцию:

fscanf(f, "%s", ptr);

Эта функция работает аналогично функции scanf. Поэтому использовать эту функцию небезопасно! Все проблемы, перечисленные для scanf-а, имеют место и при работе с fscanf-ом.

Если мы хотим записать в файл что-то, то мы должны сначала открыть его на запись:

FILE \*f = fopen("file2.html", "w");

Тут "w" означает, что мы открываем файл на запись (от write). Если файл не существовал, то он создастся и откроется на запись, а если он существовал, то он сначала будет уничтожен, а затем создан заново, и потом файл будет открыт на запись.

Еще один способ открыть файл -- это открыть его на дозапись. Это можно сделать с помощью параметра "a" (от append). Если файл не существовал, то он создастся и откроется на запись, а если он существовал, то он откроется на запись, и запись будет производится в конец файла.

Затем можно использовать функцию fprintf(f, ...)

### Текстовые и бинарные (двоичные) файлы

Язык С++ унаследовал от языка C библиотеку стандартных функций ввода-вывода. Функции ввода-вывода объявлены в заголовочном файле <stdio.h>. Операции ввода-вывода осуществляются с файлами. Файл может быть текстовым или бинарным (двоичным). Различие между ними заключается в том, что в текстовом файле последовательности символов разбиты на строки. Признаком конца строки является пара символов CR (возврат каретки) и LF (перевод строки) или, что то же самое - '\г' + '\n'. При вводе информации из текстового файла эта пара символов заменяется символом CR, при выводе, наоборот, - символ CR заменяется парой символов CR и LF. Бинарный (или двоичный) файл - это просто последовательность символов. Обычно двоичные файлы используются в том случае, если они являются источником информации, не предполагающей ее непосредственного представления человеку. При вводе и выводе информации в бинарные файлы никакого преобразования символов не производится.

**Текстовые и бинарные файлы; что меняет опция t/b**

Рассмотрим строку:

fopen(f, "file1.txt", "w");

Почему второй параметр "w" является строкой, а не символом?

На самом деле бывает много способов прочитать/записать файл. Например:

fopen("file1.txt", "wt") -- откроет файл как текстовый файл;

fopen("file1.txt", "wb") -- откроет файл как бинарный файл.

Но в чем отличие? Разница заключается лишь в том, что символы переноса строк запишутся по разному. Рассмотрим пример в UNIX и Windows.

Исходная строка кода выглядит так:

fprintf("Hello\n");

1. Откроем в Windows файл на запись с параметром "wb" (как бинарный файл). Это означает, что в него запишется в точности то, что мы передали в функции fprintf. Тогда в файл запишутся ровно 6 байт:   
     
   Hello\10
2. А теперь мы откроем в Windows файл на запись с параметром "wt" (как текстовый файл). Тогда в файл запишутся ровно 7 байт:  
     
   Hello\10\13  
     
   Тут \10\13 означает символы перевода строки в ОС Windows.
3. Откроем в UNIX файл на запись с параметром "wt" или "wb". Тогда в файл запишутся ровно 6 байт:  
     
   Hello\10  
     
   Тут \10 означает символы перевода строки в ОС UNIX.

В ОС UNIX разницы все-таки нет.

**Различие между "wt" и "wb" объясняется тем, что в разных операционных системах символы перевода строки разные. При чтении файла, т.е. при открытии файла с параметрами "rt" или "rb", проблема следующая. Если мы поставим параметр "rb", то при чтении файла символ \10 будет восприниматься как перевод строки. А если поставим параметр "rt", то при чтении файла пара символов \10\13 будет восприниматься как символ перевода строки.**

http://rsdn.org/forum/cpp/2277645.hot

**Двоичные файлы**

Двоичные и текстовые файлы — это, как говорят в Одессе, «две большие разницы». Обмен данными между программой и двоичным потоком выполняется без всякого преобразования, поэтому работает быстрее. Двоичный ввод выполняется функцией fread(), имеющей следующий прототип:

size\_t fread(void **\*buffer, size\_t size, size\_t n, FILE \*stream);**

Тип size\_t обычно определен как unsigned int. Первый параметр часто определяет массив (или указатель на динамический массив), в который будет прочитана информация; третий параметр задает размер одного элемента данных в байтах, а второй — количество читаемых элементов. Четвертый параметр определяет двоичный файл, из которого информация вводится. Общее количество считанных байтов равно size\*n. Однако возвращает функция количество корректно прочитанных элементов, а не байтов.

Вывод в двоичный файл выполняется функцией fwrite(), которая имеет совершенно аналогичный прототип:

size\_t fwrite(constvoid **\*buffer, size\_t size, size\_t n, FILE \*stream);**

Функция записывает n элементов размера size в двоичный файл stream из буфера, указатель на который задается в качестве первого аргумента. Общее количество выводимых байтов равно size\*n. Однако возвращает функция количество корректно записанных элементов, а не байтов.

В качестве элементов могут использоваться любые переменные любых типов, в том числе и динамические. Даже массив может быть одним-единственным элементом! Рассмотрим несколько простых примеров, аналогичных примерам для текстовых файлов. Создадим на диске C: каталог BinFiles и все двоичные файлы будем размещать в нем. Переделаем пример создания файла

//Создание и чтение двоичных файлов

#include<cstdio>

#include<cstdlib>

#include<ctime>

int **main()**

{ int **m[10]={0};**

**srand((**unsigned)time(NULL)); // инициализация датчика случайных чисел

**FILE \*stream;**

/\* открываем двоичный файл для записи \*/

//if((stream = fopen("c:/binfiles/number1.bin", "wb" )) == NULL)

if((stream = fopen("number1.bin", "wb" **)) == NULL)**

return **1;** // ошибка при открытии

/\* заполняем массив m числами \*/

for(int **i = 0; i < 10; i++)**

**m[i] = rand()%10;** // случайные числа от 0 до 9

// заполняем файл number1.bin элементами-числами

for(int **i = 0; i < 10; i++)**

**fwrite(&m[i], 1,** sizeof(int), stream);

**fclose(stream);** // закрываем файл

/\* открываем другой файл для записи \*/

//if((stream = fopen("c:/binfiles/number2.bin", "wb" )) == NULL)

if((stream = fopen("number2.bin", "wb" **)) == NULL)**

return **1;** // ошибка при открытии

// заполняем файл number2.bin элементом-массивом

**fwrite(m, 1,** sizeof(m), stream); // массив - один элемент

**fclose(stream);** // закрываем файл

// вывод второго двоичного файла на экран

// открываем файл для чтения

//if((stream = fopen("c:/binfiles/number2.bin", "rb" )) == NULL)

if((stream = fopen("number2.bin", "rb" **)) == NULL)**

return **1;** // ошибка при открытии

int **a = 0;** // сюда вводим

// читаем второй файл поэлементно

// правильный цикл

**printf(**"number2.bin\n");

**fread(&a, 1,** sizeof(int), stream); // предварительное чтение

while(!feof(stream)) // пока не конец файла

**{ printf(**"%d\n", a);

**fread(&a, 1,** sizeof(int), stream);

**}**

**fclose(stream);**

// открываем первый файл в режиме чтения

**printf(**"number1.bin\n");

//if((stream = fopen("c:/binfiles/number1.bin", "rb" )) == NULL)

if((stream = fopen("number1.bin", "rb" **)) == NULL)**

return **1;**

int **t[10] = {0};** // массив для чтения

// читаем первый файл как массив

**fread(t, 1,** sizeof(t), stream);

for(int **i = 0; i < 10; i++)** // выводим поэлементно

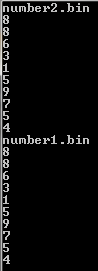
**printf(**"%d\n",t[i]);

**fclose(stream);**

char **ch = getchar();**

return **0;**

}



## Потоковые функции.

Для того чтобы использовать потоковые функции, в программу должен быть включен файл stdio.h. В этом файле содержатся описания констант, типов и структур, используемых в потоковых функциях, а также — прототипы и макроопределения этих функций.

Многие константы, предопределенные в файле stdio.h, могут быть полезны для вашей программы. Например: EOF определяется как значение, возвращаемое функциями ввода при обнаружении конца файла; NULL определяется как указатель на null. Кроме того: FILE определяет структуру используемую для хранения информации о потоке, a BUFSIZ определяет размер по умолчанию буферов потока.

### Потоковый ввод-вывод. Стандартные потоки

Термин поток происходит из представления процесса ввода-вывода в файл в виде последовательности или потока байтов. Над потоком можно выполнять следующие операции:

* Считывание блока данных из потока в оперативную память;
* Запись блока данных из оперативной памяти в поток;
* Обновление блока данных в потоке;
* Считывание записи из потока;
* Занесение записи в поток.

Вес потоковые функции ввода-вывода обеспечивают буферизированный, форматированный или не форматированный ввод и вывод.

Когда начинается выполнение программы, автоматически открываются следующие потоки:

* stdin - стандартное устройство ввода;
* stdout - стандартное устройство вывода;
* stderr - стандартное устройство сообщений об ошибках;
* stdprn - стандартное устройство печати;
* stdaux - стандартное вспомогательное устройство.

Все они называются стандартными (или предопределенными) потоками ввода-вывода. По умолчанию стандартным устройством ввода, вывода и сообщений об ошибках является пользовательский терминал. Поток стандартного устройства печати относится к принтеру, а поток стандартного вспомогательного устройства - к вспомогательному порту компьютера. По умолчанию при открытии все стандартные потоки, за исключением потоков stderr и stdaux, буферизуются.

В современных операционных системах клавиатура и дисплей рассматриваются как файлы (притом текстовые!), поскольку с клавиатуры может считываться информация в программу, а на дисплей - выводиться. При запуске программы на выполнение ей можно переназначить стандартное устройство ввода (клавиатуру) или стандартное устройство вывода (дисплей), назначив вместо этих устройств текстовый файл. При этом говорят, что происходит переадресация ввода или, соответственно, вывода. Для переназначения ввода используется символ "<", а для переназначения вывода - символ ">". Если выполняемая программа называется example.exe, следующая строка используется для переназначения ввода с клавиатуры на файл sample.dat:

example < sample.dat

Переназначение вывода с дисплея на вывод в файл осуществляется с помощью следующей строки:

example > output.dat

Наконец, следующая строка осуществляет одновременное переназначение ввода и вывода:

example < sample.dat > output.dat

Можно также осуществить соединение выходного потока одной программы с входным потоком другой. Это называется конвейерной пересылкой. Если имеются две выполнимые программы example1 и example2, то конвейерная пересылка между ними организуется с помощью символа вертикальной черты ‘|’. Следующая строка организует конвейерную пересылку между example1 и example2:

example1 | example2

Организацию конвейерной пересылки обеспечит операционная система.

### Открытие потоков.

Перед тем, как выполнять операции ввода и вывода в файловый поток, нужно его открыть с помощью **функции fopen().** Эта функция имеет следующий прототип:

FILE \*fopen(const char \*filename, const char \*mode);

Она открывает файл с именем filename и связывает с ним поток. Функция fopen() возвращает указатель, используемый для идентификации потока в последующих операциях. Параметр mode является строкой, задающей режим, в котором открывается файл. Он может принимать значения, указанные в таблице.

**Режимы открытия файла**

|  |  |
| --- | --- |
| **Значение** | **Описание** |
| R | Файл открывается только для чтения |
| W | Файл создается для записи. Если файл с этим именем уже существует, он будет перезаписан. |
| A | Режим добавления записей (Append): файл открывается для записи в конец (начиная с EOF) или создается для записи, если он еще не существует. |
| r+ | Существующий файл открывается для обновления (считывания и записи). |
| w+ | Создается новый файл для обновления (считывания и записи). Если файл с этим именем уже существует, он будет перезаписан. |
| a+ | Файл открывается для добавления (т.е. записывания, начиная с EOF): если файл еще не существует, он создается. |

Чтобы указать, что данный файл открывается или создается как текстовый, добавьте символ t в строку режима открытия (например, rt, w+t и т.д.). Аналогичным образом, чтобы сообщить, что файл открывается или создается как бинарный, добавьте в строку режима открытия символ b (например, wb, a+b и т.д.) Функция fopen() также позволяет вставить символы t или b между буквой и символом (+) в строке режима открытия (например, строка rt+ эквивалентна строке r+t). Когда файл открывается для обновления, можно вводить и выводить данные в результирующий поток. Однако вывод не может осуществляться непосредственно после ввода, если ему не предшествует вызов функции fseek() или rewind(). В случае успеха fopen() возвращает указатель на открытый поток; в случае ошибки — указатель NULL.

Например:

FILE\* stream = fopen(“Install.dat”, “r”);

Указатель на открытый файловый поток используется во всех последующих функциях работы с потоком.

Перед тем как выполнять операции ввода и вывода для потока, можно с помощью одной из трех функций открыть этот поток: fopen(), fdopen() или freopen(). В момент открытия потока задаются режим файла и способ доступа. Файл потока может открываться для считывания, записи или для считывания/записи в текстовом или двоичном режиме.

Все три функции возвращают указатель на файл, который используется для обращения к потоку. Например: после выполнения следующего оператора можно использовать указатель на файл pfinfile для ссылки на поток:

pfinfile = fopen("input.dat","r");

Когда начинается выполнение приложения, автоматически открываются пять следующих потоков: стандартное устройство ввода (standard input — stdin), стандартное устройство вывода (standard output — stdout), стандартное устройство сообщений об ошибках (standard error — stderr), стандартное устройство печати (standard printer — stdprn) и стандартное вспомогательное устройство (standard auxiliary — stdaux).

Эти пять файловых указателей можно использовать во всех функциях, которые имеют в качестве параметра указатель на поток. Некоторые функции, например getchar() и putchar(), автоматически используют stdin или stdout. Поскольку указатели stdin, stdout, stderr, stdprn и stdaux являются константами, а не переменными, нельзя присвоить им новое значение указателя на поток.

### Файлы.

Чтобы показать элементарные примеры использования файлов, мы составили небольшую программу, которая читает содержимое файла, названного test, и выводит его на экран. Вы найдете наши пояснения сразу после программы.

/\* расскажите, что находится в файле "test" \*/

#include <stdio.h>

void main()

{

FILE \*in; /\* описываю указатель на файл \*/

int ch;

if ( (in = fopen("test.txt" , "r")) != NULL)

/\* открываю test для чтения, проверяя, существует ли он \*/

/\* указатель FILE ссылается теперь на test \*/

{

while ( (ch = getc(in) )!= EOF) /\* получаю символ из in \*/

putc(ch, stdout); /\* посылаю на стандартный вывод \*/

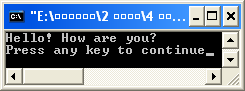
fclose(in); /\* закрываю файл \*/

}

else

printf("I couldn't open file \"test\" \n");

}



Функцией fopen() управляют три основных параметра. Первый — имя файла, который следует открыть. Он является и первым аргументом fopen(); в нашем случае это " test" .

Второй параметр [и второй аргумент fopen()] описывает, как должен использоваться файл. Вот три основных применения файла:

"r" : файл нужно считать

"w\* : файл нужно записать

"а" : файл нужно дополнить

Некоторые системы предоставляют еще дополнительные возможности, но мы будем придерживаться этих. Заметим, что используемые нами коды являются строками, а не символьными константами; следовательно, они заключаются в двойные кавычки. При применении "r" открывается существующий файл. При двух других применениях тоже будет открываться существующий файл, но если такого файла нет, он будет создан.

Внимание: Если вы используете "w" для существующего файла, то старая версия его стирается, и ваша программа начинает на "чистом месте".

Третий параметр является указателем на файл; это значение возвращается функцией:

FILE \*in;

in = fopen("test", "r");

Теперь in является указателем на файл "test". С этого момента программа ссылается на файл при помощи указателя in, а не по имени test.

Если вы очень сообразительны, то теперь можете задать такой вопрос: «Если fopen() возвращает указатель на ' FILE' в качестве аргумента, то почему мы не должны объявить fopen() как функцию указателя на ' FILE' ?» Хороший вопрос. Ответ заключается в том, что это описание сделал для нас файл stdio.h, который содержит строку

FILE \*fopen();

Есть еще один важный момент относительно функции fopen(), которую мы использовали. Если fopen() не способна открыть требуемый файл, она возвращает значение 'NULL' (определенное в stdio.h как 0). Почему она не смогла открыть файл? Вы можете попросить ее считать файл, который не существует. Вот почему мы имеем в программе строку

if ( (in = fopen("test", "r") )!= NULL)

Заполнение диска, использование запрещенного имени или некоторые другие причины могут препятствовать открытию файла. Поэтому побеспокойтесь, чтобы их не было — маленькая ошибка может увести вас очень далеко.

Закрыть файл проще.

Рано или поздно в программе потребуется ввод или вывод не через клавиатуру или дисплей, а непосредственно из файлов. В следующем примере показано, как объявлять и использовать простые файлы данных:

/\*В данной программе на С показано, как объявлять и использовать

файлы для ввода и вывода. Программа считывает значение order\_price

из файла customer.txt и вычисляет значение billing\_price,

записываемое в файл billing.txt\*/

#include <stdio.h>

#define MIN\_DISCOUNT 0.97

#define MAX\_DISCOUNT 0.95

main ()

{

float forder\_price, fbilling\_price;

FILE \*fin,\*fout;

fin=fopen ("customer.txt", "r") ;

fout=fopen ("billing.txt", "w") ;

while (fscanf(fin,"%f",&forder\_price) != EOF) {

fprintf(fout,"Your order of \t\t$%8.2f\n", forder\_price);

if (forder\_price < 10000)

fbilling\_price = forder\_price \* MIN\_DISCOUNT;

else fbilling\_price = forder\_price \* MAX\_DISCOUNT;

fprintf(fout,"is discounted to \t$%8.2f.\n\n",

fbilling\_price);

}

return(0);

}

Каждый файл в программе на С должен быть связан с некоторым указателем файла (file pointer). Указатель файла указывает на информацию, описывающую различные атрибуты файла, включая путь к файлу, его имя и статус. Указатель файла представляет из себя переменную-указатель типа FILE, описанную в stdio.h. В данном операторе, взятом из нашего примера, объявляется два файла \*fin и \*fout:

FILE \*fin, \*fout;

Следующие операторы программы открывают два различных потока и связывают каждый файл с соответствующим потоком:

fin=fopen ("a: \\customer.dat", "r") ;

fout=fopen("a:\\billing.dat", "w");

При переходе от чтения к записи и наоборот не забывайте позиционировать указатель файла при помощи fsetpos(), fseek() или rewind()

Сам С автоматически закрывает файлы при окончании программы. Иногда, однако, требуется самим управлять закрытием файлов. В следующем листинге приведена та же программа, но с включением необходимых вызовов функций закрытия файлов:

/\*В данной программе на С показано, как объявлять и использовать

файлы для ввода и вывода. Программа считывает значение order\_price

из файла customer.txt и вычисляет значение billing\_price,

записываемое в файл billing.txt\*/

#include <stdio.h>

#include<iomanip.h>

#define MIN\_DISCOUNT .97

#define MAX\_DISCOUNT .95

main ()

{

float forder\_price, fbilling\_price;

FILE \*fin,\*fout;

fin=fopen ("customer.txt", "r") ;

fout=fopen ("billing.txt", "w") ;

while (fscanf(fin,"%f",&forder\_price) != EOF) {

fprintf(fout,"Your order of \t$%8.2f\n", forder\_price);

if (forder\_price < 10000)

fbilling\_price = forder\_price \* MIN\_DISCOUNT;

else fbilling\_price = forder\_price \* MAX\_DISCOUNT;

fprintf(fout,"is discounted to \t$%8.2f.\n\n", fbilling\_price);

}

fclose(fin);

fclose(fout);

return(0);

}

Следующая программа выполняет те же функции, что и предыдущая, но на языке C++:

// В данной программе на C++ показано, как объявлять и использовать

// файлы для ввода и вывода. Программа считывает значение order price

// из файла customer.dat и вычисляет значение billing\_price,

// записываемое в файл billing.dat

#include <fstream.h>

#include <stream.h>

#include <iomanip.h>

#define MIN\_DISCOUNT 97

#define MAX\_DISCOUNT 95

main()

{

float forder\_price, fbilling\_price;

lfstream fin ("a:\\customer.dat") ;

of stream fout ("a:\\billmg.dat") ;

{ fin >> forder\_price;

while (!fin.eof()) {

fout << setiosflags(ios::fixed);

fout << "Your order of \t\t$" << setprecision \

<< setw(8) << forder\_price << "\n";

if (forder\_price < 10000)

fbilling\_price = forder\_price \* MIN\_DISCOUNT;

else fbilling\_price = forder\_price \* MAX\_DISCOUNT;

fout << "is discounted to \t$" << setprecision \

<< setw(8) << fbilling\_price << ".\n\n";

fin >> forder\_price;

}

fin. close ();

fout.close();

return(0);

}

Ввод/вывод дисковых файлов в C++ немного отличается от соответствующего ввода/вывода в С. В C++ имеются два средства в библиотеке потоков: объект streambuf и потоки. Одна и та же модель выполняет ввод/вывод как для клавиатуры и терминала, так и для диска. Одни и те же операторы и операции выполняются аналогичным образом. Это значительно упрощает программирование, в котором всегда были сложности и разночтения. Для упрощения дискового файлового ввода/вывода в библиотеке потоков описан объект filebuf, производный от стандартного типа streambuf. Так же как и родительский класс, объект filebuf управляет буфером, только в последнем случае буфер связан с дисковым файлом.

### Текстовые файлы с буферизацией.

**Важность буфера при работе с файлами**

Рассмотрим следующую ситуацию. Программа пишет протокол своих действий в файл (например, с помощью функции fprintf). Допустим, что программа сломалась. Понятно, что скорее всего получится так, что в файл последний fprintf (последний протокол действий) не запишется. Причина тому -- это буфер. Чтобы "протолкнуть" буфер в файл, используется функция

fflush(f)

В коде это выглядит примерно так:

fprintf(file, "%d", data); //данные записались в буфер

fflush(file); //данные из буфера "проталкнулись" в файл

Функции fopen() и fclose() работают с текстовыми файлами с «буферизацией». Под буферизацией мы понимаем, что вводимые и выводимые данные запоминаются во временной области памяти, называемой буфером. Если буфер заполнился, содержимое его передается в блок, и процесс буферизации начинается снова. Одна из основных задач fclose() заключается в том, чтобы «освободить» любые частично заполненные буфера, если файл закрыт.

Текстовым считается файл, в котором информация запоминается в виде символов в коде ASCII (или аналогичном). Он отличается от двоичного файла, который обычно используется для запоминания кодов машинного языка.

Функции ввода-вывода, о которых мы собираемся рассказать, предназначены только для работы с текстовыми файлами.

### Переназначение ввода и вывода.

Современные операционные системы рассматривают клавиатуру и видео-дисплей как файлы. Это имеет смысл постольку, поскольку система может читать с клавиатуры так же, как она читает из дискового или ленточного файла; то же можно сказать и о видеодисплее.

Предположим, что программа читает с клавиатуры и пишет на дисплей. Допустим, что вы хотите, чтобы ввод шел из файла с именем SAMPLE.DAT. Ту же самую программу можно использовать, если указать системе заменить ввод с клавиатуры, рассматриваемой как файл, на ввод из другого файла, а именно — файла SAMPLE.DAT. Процесс изменения стандартного устройства ввода или вывода называется переназначением (или переадресацией — redirection) ввода или вывода. Переназначение ввода и вывода не представляет трудностей. Для переназначения ввода используется символ <, а для переназначения вывода >. Допустим, что выполняемая программа называется REDIRECT. Следующая системная команда запускает программу REDIRECT и использует для ввода вместо клавиатуры файл SAMPLE.DAT:

redirect < sample.dat

В следующей команде переназначаются и ввод (SAMPLE.DAT), и вывод (SAMPLE.BAK):

redirect < sample.dat > sample.bak

Последняя команда переназначает только вывод (SAMPLE.BAK):

redirect > sample.bak

Заметим, однако, что стандартный файл ошибок STDERR переназначать нельзя.

Имеются два метода управления связью между стандартным файлом и физическим файлом или устройством: переназначение и конвейерная пересылка. Конвейерной пересылкой называется непосредственное соединение стандартного вывода одной программы со стандартным вводом другой программы. Управление переназначением и конвейерной пересылкой обычно происходит вне программы; эта задача и ставится, так как в этом случае самой программе не нужно знать, откуда реально поступают данные и куда пересылаются.

Для того чтобы соединить стандартный вывод одной программы со стандартным вводом другой программы, нужно назначить между ними конвейерную пересылку при помощи символа вертикальной черты "|". Следовательно, для связи стандартного вывода программы PROCESS 1 со стандартным вводом программы PROCESS2 достаточно указать:

Process1 | process2

Операционная система обеспечит все операции физической пересылки данных с выхода PROCESS 1 на вход PROCESS2.

### Изменение буфера потока.

По умолчанию все файлы, открытые при помощи потоковых функций (stdin(), stdout() и stdprn()), буферизируются. Исключение составляют открытые потоки stderr и stdaux, которые по умолчанию не буферизируются, кроме случаев, когда они используются с группой функций printf() или scanf() — тогда им назначается временный буфер. Можно задать буферизацию потоков stderr и stdaux при помощи функций setbuf() или setvbuf(). Потоки stdin, stdout и stdprn очищаются автоматически при их заполнении. При помощи двух функций setbuf() и setvbuf() можно сделать буферизированный поток небуферизированным и наоборот. Обращаем внимание на то, что буферы, выделяемые системой, недоступны для пользователя, а буферы, которые назначаются при помощи функций setbuf() и setvbuf(), именуются пользователем, и с ними можно работать как с переменными. Буферы потоков, определяемые пользователем, очень полезны для проверки ввода и вывода до того, как возникнут какие-либо системные ошибки.

Можно назначить буфер любого размера; при использовании функции setbuf() размер определяется константой BUFSIZ, описанной в файле stdio.h. Синтаксис функции setbuf() выглядит так:

void setbuf(FILE \*stream, char \*buffer);

В следующем примере setbuf() и BUFSIZ используются для описания и назначения буфера потоку stderr. Буферизация stderr предоставляет программе больше возможностей для управления обработкой исключительных ситуаций. Выполните пошаговую трассировку начального варианта программы при помощи встроенного отладчика.

/\*Программа на С, демонстрирующая описание и назначение буфера

потоку stderr\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming\AlgorithmProgramming 02\Universal\_HederFile.h"

void StopWait(void);

char cmyoutputbuffer[BUFSIZ];

main(void)

{

/\* связь буфера с небуферизированным выходным потоком \*/

setbuf(stderr, cmyoutputbuffer); /\* строка, которую нужно \*/ /\*ЗАКАМЕНТИРУЙ для 2 варианта\*/

/\* заполнение буфера выходного потока \*/

fputs("Sample output inserted into the\n",stderr); /\*"Пример вывода в" \*/

fputs("output stream buffer.\n",stderr); /\* "буфер выходного потока." \*/

/\* вывод буфера выходного потока \*/

fflush(stderr);

StopWait(); /\* Wait a little \*/

return (0);

}

Попробуйте запустить программу второй раз, закомментировав оператор setbuf(). В этом случае программа не будет связывать буфер с stderr. Вы увидите различие после запуска программы? Когда stderr не буферизируется, встроенный отладчик выдает сообщение каждого оператора fputs() сразу же после выполнения оператора.

В следующей программе используется функция setvbuf(); ее синтаксис выглядит следующим образом:

int setvbuf (FILE \*stream, char \*buffer, int buftype, size\_t bufsize);

В этом примере размер буфера задается явно, а не при помощи константы BUFSIZ, описанной в файле stdio.h:

/\*Программа на С, демонстрирующая использование функции setvbuf()\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming\Universal\_HederFile.h"

void StopWait(void);

#define MYBUFSIZ 512

main (void)

{

char ichar, cmybuffer[MYBUFSIZ];

FILE \*pfinfile, \*pfoutfile;

pfinfile = fopen("E:\\LECTURE\\AlgorithmProgramming\\sample.in", "r");

pfoutfile = fopen("E:\\LECTURE\\AlgorithmProgramming\\sample.out", "w");

if (setvbuf(pfinfile, cmybuffer, \_IOFBF, MYBUFSIZ) != 0)

printf("pfinfile buffer allocation error\n"); /\* ошибка назначения \*/

else /\* буфера \*/

printf("pfinfile buffer created\n"); /\* буфер создан \*/

if (setvbuf(pfoutfile, NULL, \_IOLBF, 132) != 0)

printf("pfoutfile buffer allocation error\n"); /\* ошибка назначения\*/

else

/\* буфера \*/

printf("pfoutfile buffer created\n");

/\* буфер создан \*/

while(fscanf(pfinfile,"%c",&ichar) != EOF)

fprintf(pfoutfile,"%c",ichar);

fclose(pfinfile);

fclose(pfoutfile);

StopWait(); /\* Wait a little \*/

return (0);

}

В программе создается доступный пользователю буфер, на который указывает pfinfile, и автоматически выделенный (malloc()) буфер, на который указывает pfoutfile. Последний из двух буферов описан как buftype, \_IOLBF или строчный буфер. Другие опции, описанные в файле stdio.h: \_IOFBF — полная буферизация и \_IONBF — отсутствие буферизации. Напоминаем, что при использовании функций setbuf() и setvbuf() вместо автоматически выделяемого буфера для ввода/вывода используется buffer, назначенный пользователем. Если в случае использования setbuf() пара-метр buffer равен null, то ввод/вывод будет небуферизированным. В противном случае он буферизируется полностью.

Если в случае использования setvbuf() параметр buffer равен null, то буфер выделяется при помощи функции malloc(). При создании buffer функция setvbof() использует параметр bufsize, который определяет количество выделяемой и автоматически освобождаемой при закрытии памяти.

### Закрытие потоков.

По завершении работы с потоком он должен быть закрыт. Это осуществляется с помощью функции fclose(), которая имеет следующий прототип:

int fclose(FILE \*stream);

Все буферы, связанные с потоком, освобождаются перед закрытием потока. В случае успеха fclose() возвращает 0; в случае ошибки - EOF. Если ваша программа не закрывает поток с помощью явного вызова fclose (), то он закрывается автоматически по ее завершению.

Две функции fclose() и fcloseall() используются для закрытия потока или всех потоков, соответственно. Функция fclose() закрывает отдельный файл, а функция fcloseall() закрывает все открытые потоки, за исключением stdin, stdout, stderr, stdprn и stdaux. Однако, если ваша программа не закрывает поток явно, то он закрывается автоматически по ее завершению. Поскольку количество потоков, открытых одновременно, ограничено, то желательно закрывать поток по окончанию работы с ним.

Наш пример показывает, как закрывать файл:

fclose(in);

Просто используйте функцию fclose(). Заметим, что аргументом ее является in, указатель на файл, а не test, имя файла.

Для программы, более серьезной, чем эта, следовало бы посмотреть, успешно ли закрыт файл. Функция fclose() возвращает значение 0, если файл закрыт успешно, и — 1 в противном случае.

**Зачем нужно закрывать файлы**

* Зададимся вопросом: "Что надо сделать после того, как мы поработали с файлом?"

Формальный ответ: "Закрыть файл." Это можно сделать с помощью функции:

fclose(f);

Но зачем это делать?

Ввиду механического устройства жесткого диска, данные в файл попадают не сразу. Сначала данные записываются в так называемый буфер (область оперативной памяти), и когда он переполнится, то данные из буфера будут записаны в файл. Такая схема придумана для ускорения работы с файлами. На самом деле, буфер -- это поле структуры FILE: указатель на массив char-ов.

Если мы напишем fprintf(...), то запись произведется в буфер. И только тогда, когда буфер будет заполнен до конца, он будет сразу весь записан на жесткий диск. По этой причине, если мы не закроем файл функцией fclose(f), то последние данные из буфера не запишутся в файл. Отсутствие этой команды может привести к потере данных в файле, который был открыт для записи (дозаписи).

* А зачем закрывать файлы, открытые только на чтение?

Если не закрывать файлы (которые открыты даже для чтения), то это может привести к ограничению доступа к файлу для других программ. Какие именно ограничения наложатся - это зависит от ОС. Но в ОС Windows если файл открыт на чтение и не закрывается, то из другой программы его нельзя удалить.

* В любой ОС есть ограничение на количество одновременно открытых файлов. И это еще одна причина для закрытия файлов.

### Функции, осуществляющие ввод-вывод в файловый поток

Рассмотрим теперь функции, осуществляющие ввод-вывод в файловый поток.

**Функция fgetc()** имеет следующий прототип:

int fgetc(FILE \*stream);

Она осуществляет ввод символа из файлового потока stream. В случае успеха функция преобразует прочитанный символ в тип int без учета знака. Если делается попытка прочесть конец файла или произошла ошибка, функция возвращает EOF. Как видим, эта функция во всем аналогична функции getc(), за исключением того, что чтение осуществляется из файлового потока. Более того, как мы уже отмечали, на самом деле getc() — это макрос, реализованный с помощью fgetc(). То же самое относится и к следующим функциям: все они имеют уже рассмотренные нами аналоги.

**Функция fputc()** имеет следующий прототип:

int fputc(int с, FILE \*stream);

Она осуществляет вывод символа в файловый поток и во всем аналогична функции putc().

**Функция fgets()** имеет следующий прототип:

char \*fgets(char \*s, int n, FILE \*stream);

Она осуществляет чтение строки символов из файлового потока в строку s. Функция прекращает чтение, если прочитано n — 1 символов или встретился символ перехода на новую строку ‘\n’. Если встретился символ перехода на новую строку, он сохраняется в переменной s. В обоих случаях в переменную s добавляется символ ‘\0’, который знаком завершения строковой переменной, функция возвращает строку, на которую указывает параметр s. Если делается попытка чтения конца файла или произошла ошибка, она возвращает NULL.

**Функция fputs()** имеет следующий прототип:

int fputs(const char \*s, FILE \*stream);

Она осуществляет вывод строки в файловый поток. Символ перехода на новую строку не добавляется, и завершающий строку нуль-символ в файловый поток не копируется. В случае успеха fputs() возвращает неотрицательное значение. В противном случае она возвращает EOF.

**Функция fscanf()** имеет следующий прототип:

int fscanf(FILE \*stream,

const char \*format[, address,...]);

Она во всем аналогична функции scanf(), за исключением того, что форматированный ввод осуществляется не со стандартного устройства ввода, а из файлового потока.

**Функция fprintf()** имеет следующий прототип:

int fprintf(FILE \*stream,

const char \*format[, argument,...]);

Она во всем аналогична функции printf(), но осуществляет форматированный вывод не на стандартное устройство вывода, а в файловый поток.

**Функция feof()** является на самом деле макросом и позволяет осуществлять проверку на достижение символа конца файла при операциях ввода-вывода. Она имеет следующий прототип.

int feof(FILE \*stream);

Она возвращает ненулевое значение, если был обнаружен конец файла при последней операции ввода в поток stream и 0, если конец файла еще не достигнут.

Рассмотрим пример файлового ввода и вывода:

#include <stdio.h>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

FILE \*in, \*out;

if ((in = fopen("C:\\AUTOEXEC. BAT", "rt")) == NULL){

fprintf(stderr,

"Cannot open input file.\n");

getchar(); getchar();

return 1;

}

if ((out = fopen("C:\\AUTOEXEC.BAK", "wt")) == NULL){

fprintf(stderr,

"Cannot open output file.\n");

getchar(); getchar();

return 1;

}

while (!feof(in))

fputc(fgetc(in), out);

fclose(in);

fclose(out);

fprintf(stderr,

"The file is copied successfully.\n");

getchar(); getchar();

return 0;

}



Две следующие функции предназначены для осуществления неформатированного ввода и вывода в файловые потоки. **Функция fread()** имеет следующий прототип:

size\_t

fread(void \*ptr, size\_t size,

size\_t n, FILE \*stream);

Эта функция считывает из потока stream в буфер, указанный параметром ptr, n блоков данных, каждый из которых содержит size байтов. В случае успеха функция возвращает число прочитанных блоков. Если прочитан конец файла или произошла ошибка, она возвращает число полностью прочитанных блоков или 0.

**Функция fwrite()** имеет следующий прототип:

size\_t

fwrite(const void \*ptr, size\_t size,

size\_t n, FILE \*stream);

Она записывает в выходной поток stream из буфера, указанного параметром ptr, n блоков данных, каждый из которых содержит size байтов. В случае успеха функция возвращает число записанных блоков. В случае ошибки она возвращает число полностью записанных блоков или 0.

Приведем пример, демонстрирующий работу с этими функциями:

#include <string.h>

#include <stdio.h>

using namespace std;

struct Client{

int Num;

char SurName[27];

char Name[21];

char SecName[21];

};

int main(void)

{

setlocale(LC\_ALL, "Rus");

FILE \*stream;

Client AClient, RClient;

//Открываем файл для вывода

if ((stream = fopen("SAMPLE.DAT", "wb"))== NULL){

printf("Cannot open output file.\n");

getchar(); getchar();

return 1;

}

AClient.Num = 1;

strcpy(AClient.SurName,"Petrov");

strcpy(AClient.Name, "Petr");

strcpy(AClient.SecName, "Petrovich");

//Запись структуры в файл

fwrite(&AClient, sizeof(AClient), 1, stream);

//Закрываем файл

fclose(stream);

//Открываем файл для чтения

if ((stream = fopen("SAMPLE.DAT", "rb")) == NULL){

printf("Cannot open input file.\n");

getchar(); getchar();

return 2;

}

fread(&RClient, sizeof(RClient), 1, stream);

//Закрываем файл fclose(stream);

printf("The structure contains:\n");

printf("Num = %d SurName = %s "

"Name = %s SecName = %s",

RClient.Num, RClient.SurName,

RClient.Name, RClient.SecName);

getchar(); getchar();

return 0;

}

В этом примере вначале файл открывается для записи в него значений полей из структуры AClient, затем этот файл открывается для чтения содержащейся в нем информации в структуру RClient. После чего поля этой структуры выводятся на экран. При выполнении программа выводит на экран:

The structure contains:

Num = 1 SurName = Petrov Name = Petr

SecName = Petrovich



## Низкоуровневый ввод и вывод в С.

В следующем списке перечислены наиболее часто используемые в про-граммах функции низкоуровневого ввода и вывода:

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| close() | Закрывает дисковый файл |
| lseek() | Поиск указанного байта в файле |
| open() | Открывает дисковый файл |
| read() | Читает данные в буфер |
| unlink() | Удаляет файл из подкаталога |
| write() | Записывает буфер данных |

Функции низкоуровневого ввода и вывода не буферизируют и не форматируют данные. Доступ к файлам, открываемых функциями нижнего уровня, осуществляется при помощи описателя файла (handle — целое число, используемое операционной системой для обращения к файлу). Для открытия файлов используется функция ореn(). Для открытия файла с атрибутами совместного использования (sharing) можно использовать макрос sopen().

Низкоуровневые функции отличаются от своих аналогов — потоковых функций — тем, что они не требуют включения заголовочного файла stdio.h. Однако, могут быть полезны некоторые часто используемые константы, предопределенные в stdio.h: например, EOF и NULL. Объявления низкоуровневых функций дается в заголовочном файле io.h.

Первоначально данная система дискового ввода/вывода была создана для операционной системы UNIX. Поскольку комитет по стандарту ANSI С решил не стандартизовывать эту UNIX-подобную систему небуферизированного, низкоуровневого ввода/вывода, то ее нельзя рекомендовать для дальнейшего использования. Для всех новых разработок рекомендуется использовать стандартизованную систему буферизированного ввода/вывода.

**File descriptors. Open, close, read, write**

В языке C есть много способов работы с файлами. Помимо структуры FILE можно использовать так называемые дескрипторы файла (file descriptors). Дескриптор файла -- целое неотрицательное число. Оно обозначает номер открытого файла в таблице открытых файлов операционной системы. Использование дескрипторов файла -- более низкий уровень, чем нежели ипользование струкруты FILE. Структура FILE -- сущность языка C и его стандартной библиотеки, тогда как дескриптор файла -- сущность операционной системы. Например, при работе со структурой FILE автоматически создается буфер, и программист работает с более высокоуровневой абстракцией. А при работе с дескрипторами файла программист должен позаботится о буферизации вручную.

Пример работы с дескрипторами файла довольно прост и почти в точности повторяет процесс работы со структурой FILE:

int fd = open("...");

Сходство работы с дескрипторами файла с работой со структурой FILE заключается в том, что в названии функций отсутствует буква "f". Иногда параметры функций незначительно отличаются.

Структуру FILE полезно использовать при работе с настоящими "файлами" (которые находятся на жестком диске). Ипользовать дескрипторы файла полезно в случаях работы со специальными "файлами". В этом подходе есть своя специфика работы, но сейчас просто полезно знать, что такой подход существует.

Аналогами stdin, stdout и stderr в дескрипторах файла являются числа 0, 1 и 2 соответственно. Стандарт POSIX.1 обозначил числа 0, 1, 2 символическими константами STDIN\_FILENO, STDOUT\_FILENO и STDERR\_FILENO соответственно.

**Memory mapping. Функция mmap**

Следующий способ работы с файлами удобен в тех случаях, когда приходится читать файл нелинейно: надо "ходить" вперед и назад. В предыдущих подходах такие ситуации оказывались неудобными с точки зрения программирования: получился бы громоздкий код.

В языке C был придуман удобный способ работы в таких ситуациях, который называется memory mapping. Соответствующая функция:

char \*ptr = mmap("...");

Работает эта функция примерно так. Мы указываем этой функции файл на диске, и она "отображает" этот файл в такую-то область в памяти. В результате работы функции мы получаем указатель на начало файла. И потом мы можем работать с этим файлом как с обычным указателем на какую-то область памяти: можем "ходить" вперед и назад по этому файлу.

Можно "отобразить" не весь файл целиком, а, например, отдельную часть файла: с 3-его килобайта по 4-ый килобайт.

**Win32 API: FileCreate, FileRead, etc.**

При работе с файлами в ОС Windows можно использовать все те функции, которые были описаны выше. В ОС Windows есть своя большая стандартная библиотека Win32 API. В этой библиотеке также есть функции для работы с файлами: например, функции FileCreate(...) или FileOpen(...). Они по своей работе похожи на функции из стандартной библиотеки C, но отличия также присутствуют. Они заключаются в параметрах этих функций и небольших "хитростях", которые мы здесь опустим.

Если вы программируете под ОС Windows и пишите программу для работы в ОС Windows, то стоит пользоваться библиотекой Win32 API для работы с файлами.

## Ввод и вывод символов.

Во всех компиляторах С имеются несколько описанных в стандарте ANSI С функций, предназначенных для ввода и вывода символов. Эти функции обеспечивают стандартный ввод и вывод и рассматриваются как высокоуровневые процедуры (в отличие из низкоуровневых функций, которые более непосредственно обращаются к аппаратному обеспечению компьютера). Ввод/вывод в С реализован не при помощи ключевых слов, являющихся частью языка, а с использованием функций, поставляемых разработчиками компилятора.

### Использование функций getc(), putc(), fgetc() и fputc().

Две функции getc() и putc() работают аналогично функциям getchar() и putchar(). Разница заключается в том, что вы должны сообщить новичкам, какой файл следует использовать. Таким образом, наш «старый дружище»

ch = getchar();

предназначен для получения символа от стандартного ввода, а

ch = getc(in);

— для получения символа от файла, на который указывает in. Аналогично функция

putc(ch, out);

предназначена для записи символа ch в файл, на который ссылается указатель out типа FILE. В списке аргументов функции putc() этот символ стоит первым, а затем указатель файла. В нашем примере мы использовали

putc(ch, stdout);

где stdout является указателем на стандартный вывод. Таким образом, этот оператор эквивалентен

putchar(ch);

Действительно, оператор putchar(ch) определен директивой #define так же как функция putc(ch, stdout) определена в файле stdio.h. Этот ужасный файл к тому же определяет в директиве #define указатели stdout и stdin на стандартный вывод и стандартный ввод системы.

Наиболее общими из всех функций ввода/вывода являются те, которые работают с отдельными символами. Функция getc() вводит один символ из указанного файлового потока:

int ic;

ic = getc(stdin);

Символ вводится при указании имени функции getc(), и затем возвращаемое значение присваивается переменной ic. Если же вас удивит, почему переменная ic не имеет тип char, то это объясняется тем, что функция getc() согласно прототипу возвращает тип int. Это нужно для того, чтобы можно было иметь системно-зависимый размер указателя конца файла, который может не совпадать с размером отдельного байта типа char.

Функция getc() преобразует целое число в беззнаковый символ. Использование беззнакового символа вместо целого числа гарантирует, что ASCII-символы, имеющие значение > 127, не будут представляться как отрицательные значения. Следовательно, отрицательные значения можно использовать для представления нестандартных ситуаций, например, ошибок и конца входного файла. К примеру: конец файла обычно представляется значением -1, хотя в стандарте ANSI С записано, что только константа EOF может иметь некоторое отрицательное значение.

Поскольку getc() возвращает целое число, то переменные, использующие результат функции getc(), также должны быть описаны как целые. Хотя использование целых чисел в символьной функции может показаться не-удобным, на самом деле в языке С символы и целые числа отличаются очень мало. Когда вместо целого подставляется символ, он автоматически преобразуется в целое число.

Эта группа функций ввода-вывода на самом деле реализована в виде макросов.

Макрос getc() возвращает следующий символ из заданного входного потока и увеличивает указатель входного потока так, чтобы он указывал на следующий символ. В случае успеха getc () возвращает считанный символ, преобразованный в тип int без знака. Если прочитан конец файла или произошла ошибка, он возвращает EOF. Прототип этого макроса следующий:

int getc(FILE \*stream);

Символ EOF определяется следующим образом:

#define EOF (-1)

В операциях ввода-вывода он служит для обозначения и проверки конца файла.

Подразумевается, что этот символ имеет тип signed char, еcли в операциях ввода-вывода участвуют символы типа unsigned char, то использовать EOF нельзя. Следующий пример демонстрирует использование этого макроса:

#include <stdio.h>

using namespace std;

int main ()

{

setlocale(LC\_ALL, "Rus");

char ch;

printf("Bвод символа:");

//Чтение символа из стандартного

//входного потока

ch = getc(stdin);

printf("Был введен символ: '%c'\n", ch);

getchar(); getchar();

return 0;

}



Дополнением к функции getc() служит putc(). **Макрос putc()** имеет следующий прототип:

int putc(int с, FILE \*stream);

Он выводит символ в поток, заданный параметром stream. В случае успеха putc () возвращает выводимый символ с. преобразованный к типу int; в случае ошибки - EOF.

**Макрос putchar ()** имеет следующий прототип:

int putchar(int с);

Этот макрос определяется как putc(с, stdout). В случае успеха он возвращает символ с, преобразованный к типу int. В случае ошибки он возвращает EOF.

**Функция putc()** выдает один символ в файловый поток, представленный заданным указателем файла. Для вывода введенного выше символа на стандартное устройство вывода используется следующий оператор:

putc(ic,stdout);

**Функция getc()** обычно буферизируется, и когда приложение запрашивает символ, управление не передается программе до тех пор, пока в стандартном входном файловом потоке не будет введен символ возврата каретки. Все символы, введенные до символа возврата каретки, хранятся в буфере и передаются последовательно в программу, которая повторно вызывает функцию getc() до тех пор, пока буфер не опустеет. После того, как при помощи getc() символ возврата каретки передан в программу, следующий запрос символов вызывает накопление символов в буфере до тех пор, пока опять не будет введен символ возврата каретки. Это означает, что функцию getc() нельзя использовать для обработки одиночных клавиш, когда не требуется дополнительного нажатия клавиши возврата каретки. И одно последнее замечание: на самом деле getc() и putc() являются не истинными функциями, а макросами. Функции fgetc() и fputс() идентичны их аналогам макросам getc() и putc().

Простая программа сжатия файла.

В нашем примере имя файла, который следовало открыть, было записано в программе. Мы не обязаны считаться с этим ограничением. Используя аргументы командной строки, можно сообщить нашей программе имя файла, который хотим считать. В нашем следующем примере так и происходит. С помощью, примитивного приема сжимается содержимое — остается только каждый третий символ. Наконец, сжатая версия размещается в новый файл, имя которого состоит из старого имени с добавкой .red (сокращение слова reduced). Обычно весьма важны первый и последний элементы (аргумент командной строки и добавка к имени файла). Само же сжатие имеет более ограниченное применение, но, как вы увидите, им можно пользоваться.

/\* сожмите ваши файлы в 2—3 раза! \*/

#include <stdio.h>

#include<string.h>

void main(int argc, char \*argv[])

{

FILE \*in, \*out; /\* описывает два указателя типа FILE \*/

int ch;

static char name [20]; /\* память для имени выходного файла \*/

int count = 0;

if ( argc < 2) /\* проверяет, есть ли входной файл \*/

printf("Sory, i need the title of the file.\n" );

else

{

if ( (in = fopen(argv[1], "r")) != NULL)

{

strcpy(name, argv[1]); /\* копирует имя файла в массив \*/

strcat(name, " .red"); /\* добавляет .red к имени \*/

out = fopen(name, " w"); /\* открывает файл для записи \*/

while ( (ch = getc(in)) ! = EOF)

if ( count++ %3 ==0)

putc(ch, out); /\* печатает каждый третий символ \*/

fclose(in);

fclose(out);

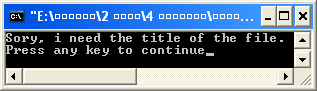
}

else

printf(" Я не смогла открыть файл\" %s\" \n", argv[1]);

}

}



Мы поместили программу в файл, названный reduce и применили эту программу к файлу, названному eddy, который содержал одну единственную строку

Даже Эдди нас опередил с детским хором.

Была выполнена команда

reduce eddy

и на выходе получен файл, названный eddy .red, который содержит

Дед спел тихо

Какая удача! Наш случайно выбранный файл сделал осмысленное сокращение.

Вот некоторые замечания по программе.

Вспомните, что argc содержит определенное количество аргументов, в число которых входит имя программного файла. Вспомните также, что с согласия операционной системы argv[0] представляет имя программы, т. е. в нашем случае reduce. Вспомните еще, что argv[1] представляет первый аргумент, в нашем случае eddy. Так как сам argv[1] является указателем на строку, он не должен заключаться в двойные кавычки в операторе вызова функции.

Мы используем argc, чтобы посмотреть, есть ли аргумент. Любые избыточные аргументы игнорируются. Помещая в программу еще один цикл, вы могли бы использовать дополнительные аргументы — имена файлов и пропускать в цикле каждый из этих файлов по очереди.

С целью создания нового имени выходного файла мы используем функцию strcpy() для копирования имени eddy в массив name. Затем применяем функцию strcat() для объединения этого имени с .red.

Программа требует, чтобы два файла были открыты одновременно, поэтому мы описали два указателя типа ' FILE'. Заметим, что каждый файл должен открываться и закрываться независимо от другого. Существует ограничение на количество файлов, которые вы можете держать открытыми одновременно. Оно зависит от типа системы, но чаще всего находится в пределах от 10 до 20. Можно использовать один и тот же указатель для различных файлов при условии, что они не открываются в одно и то же время. Мы не ограничиваемся использованием только функций getc() и putc() для файлов ввода-вывода. Далее мы рассмотрим некоторые другие возможности.

### Использование функций getchar(), putchar(),fgetchar() и fputchar().

Макрос getchar() имеет следующий прототип:

int getchar();

Этот макрос определен как getc(stdin). С помощью этого макроса из стандартного входного потока stdin считывается очередной символ и его значение преобразуется в тип int без учета знака. Если переадресация стандартного входного потока не производилась, то ввод осуществляется с клавиатуры. В противном случае ввод будет осуществляться из файла, назначенного для входного потока и указанного в командной строке при вызове программы.

На самом деле два макроса getchar() и putchar() являются особыми реализациями макросов getc() и putc(), соответственно. Они всегда связаны со стандартными устройствами ввода (stdin) и вывода (stdout). Единственный способ их использования с другими файловыми потоками — переназначить стандартный ввод или стандартный вывод вне программы.

Два приведенных выше примера можно записать иначе, используя две рассматриваемые функции:

int ic;

ic = getchar() ;

и

putchar(ic);

Так же, как и getc() и putc(), getchar() и putehar() реализованы как макросы. В случае ошибки функция putchar() возвращает значение EOF. Следующий фрагмент можно использовать для проверки ошибки вывода. Использование EOF для вывода может показаться немного сомнительным, однако это — технически корректно.

if(putchar(ic) =- EOF)

printf("An error has occurred writing to stdout"); /\* При выводе на stdout произошла ошибка \*/

Макросам getchar() и putchar() эквивалентны функции fgetchar() и fputehar().

### Использование функций getch() и putch().

Функции getch() и putch() являются полноценными низкоуровневыми функциями, тесно связанными с аппаратным обеспечением, — поэтому они не подпадают под стандарт ANSI С. В персональных компьютерах (ПК) эти функции не буферизируются; это означает, что они сразу же вводят символ, нажатый на клавиатуре. Однако, их можно переназначать, и поэтому они связаны не только с клавиатурой.

Функции getch() и putch() используются так же, как и getchar() и putchar(). Обычно программы, работающие на ПК, используют getch() для перехвата клавиш, которые игнорирует функция getchar(), например, PGUP, PGDN, HOME и END. Функция getch() видит символ, введенный с клавиатуры, сразу же после нажатия клавиши; символ возврата каретки для передачи введенного символа в программу не требуется. Эта возможность позволяет использовать getch() для обработки однократных нажатий клавиш, что невозможно с функциями getc() или getchar().

На ПК функция getch() работает совершенно по-другому, чем getc() и getchar(). Частично это объясняется тем, что в ПК легко можно распознать нажатие отдельной клавиши на клавиатуре. В других системах, подобных DEC и VAX С, аппаратура не может отслеживать отдельные нажатия клавиш. Обычно в этих системах выполняется эхо-отображение введенного символа и требуется нажатие клавиши возврата каретки, причем программа не видит нажатия этой клавиши до тех пор, пока не будут введены другие символы, и тогда при ее нажатии возвращается null-символ или десятичный ноль. Кроме того, функциональные клавиши не доступны, и при их нажатии результаты не достоверны.

## Определение строк в программе.

Символьные строки представляют один из наиболее полезных и важных типов данных языка Си. Хотя до сих пор все время применялись символьные строки, мы еще не все знаем о них. Конечно, нам уже известно самое главное: символьная строка является массивом типа char, который заканчивается нуль-символом ('\0'). Здесь мы больше узнаем о структуре строк, о том, как описывать и инициализировать строки, как их вводить или выводить из программы, как работать со строками.

Ниже представлена работающая программа, которая иллюстрирует несколько способов создания строк, их чтения и вывода на печать. Мы .используем две новые функции: gets (), которая получает строку, и puts (), которая выводит строку. (Вы, вероятно, заметили сходство их имен с функциями getchar () и putchar ().) В остальном программа выглядит достаточно привычно.

/\* работа со строками \*/

#include <stdio.h>

#define MSG " У вас, наверное, много талантов. Расскажите о некоторых."

/\* константа символьной строки \*/

#define NULL 0

#define LIM 5

#define LINLEN 81 /\* максимальная длина строки + 1 \*/

char m1[] = "Только ограничьтесь одной строкой.";

/\* инициализация внешнего символьного массива \*/

char \*m2 = " Если вы не можете вспомнить что-нибудь, придумайте.";

/\* инициализация указателя внешнего символьного массива \*/

void main ()

{

char name[LINLEN];

static char talents [LINLEN];

int i;

int count = 0;

char \*m3 = " \n Достаточно обо мне-- Как вас зовут?";

/\* инициализация указателя \*/

static char \*mytal[LIM] = {"Быстро складываю числа", "Точно умножаю", "Записываю данные", "Правильно выполняю команды" , "Понимаю язык Си"};

/\* инициализация массива строк \*/

printf(" Привет! Я Клайд, компьютер. У меня много талантов\n");

printf(" %s\n", " Позвольте рассказать о некоторых из них.");

puts(" Каковы они? Ах да, вот их неполный перечень.");

for (i = 0; i < LIM; i++)

puts(mytal[i]); /\* печатает перечень талантов компьютера \*/

puts(m3);

gets(name);

printf (" Хорошо, %s, %s\n" , name, MSG);

printf (" %s\n%s\n", m1, m2);

gets(talents);

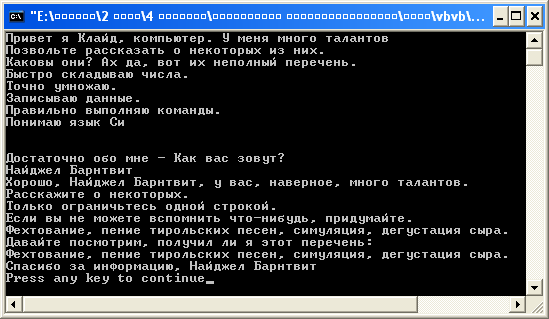
puts(" Давайте, посмотрим, получил ли я этот перечень:");

puts(talents);

printf (" Спасибо за информацию, %s.\n", name);

}

Чтобы помочь вам разобраться в том, что делает эта программа, мы приводим результат ее работы:



Тщательно исследуем программу. Но вместо того чтобы просматривать строку за строкой, применим более общий подход. Сначала рассмотрим способы определения строк в программе. Затем выясним, что нужно для чтения строки в программе. И, наконец, изучим способы вывода строки.

### Строковые константы.

Вы, вероятно, заметили, когда читали программу, что есть много способов определения строк. Попытаемся теперь рассмотреть основные: использование строковых констант, массивов типа char, указателей на тип char и массивов, состоящих из символьных строк. В программе должно быть предусмотрено выделение памяти для запоминания строки, и мы еще вернемся к этому вопросу.

Всякий раз, когда компилятор встречается с чем-то, заключенным в двойные кавычки, он определяет это как строковую константу. Символы, заключенные в кавычки, плюс, завершающий символ '\0', записываются в последовательные ячейки памяти. Компилятор подсчитывает количество символов, поскольку ему нужно знать размер памяти, необходимой для запоминания строки. Наша Программа использует несколько таких строковых констант, чаще всего в качестве аргументов функций printf () и puts (). Заметим также, что мы можем определять строковые константы при помощи директивы #define

Если вы хотите включить в строку символ двойной кавычки, ему должен предшествовать символ обратной дробной черты:

printf("\"Беги, Спот, беги!\" — сказал Дик\n");

В результате работы этого оператора будет напечатана строка:

"Беги, Спот, беги! — " сказал Дик.

Строковые константы размещаются в статической памяти. Вся фраза в кавычках является указателем на место в памяти, где записана строка. Это аналогично использованию имени массива, служащего указателем на расположение массива. Если это действительно так, то как выглядит оператор, который выводит строку?

/\* строки в качестве указателей \*/

#include <stdio.h>

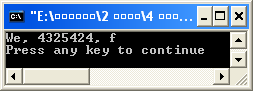
void main ()

{

printf("%s, %u, %c\n", "We", "love", \*"figs");

}

Итак, формат %s выводит строку We. Формат %u выводит целое без знака. Если слово "love" является указателем, то выдается его значение, являющееся адресом первого символа строки. Наконец, \*"figs" должно выдать значение, на которое ссылается адрес, т. е. первый символ строки "figs". Произойдет ли это на самом деле? Да, мы получим следующий текст:

****

Ну, вот! Давайте теперь вернемся к строкам, находящимся в символьных массивах.

### Массивы символьных строк и их инициализация.

При определении массива символьных строк необходимо сообщить компилятору требуемый размер памяти. Один из способов сделать это — инициализировать массив при помощи строковой константы. Так как автоматические массивы нельзя инициализировать, необходимо для этого использовать статические или внешние массивы. Например, оператор

char m1[] = "Только ограничьтесь одной строкой,";

инициализировал внешний (по умолчанию) массив m1 для указанной строки. Этот вид инициализации является краткой формой стандартной инициализации массива

char m1[] = {' Т', ' о', ' л', ' ь', ' к', ' о', ' ', 'о', 'г',

'р', 'а', 'н', 'и', 'ч', 'ь', 'т', 'е', 'с', 'ь',

' ', 'о', 'д', 'н', 'о', 'и', ", 'с', 'т',

'р', 'о', 'к', 'о', 'и', '.', '\0'

(Обратите внимание на замыкающий нуль-символ. Без него мы имеем массив символов, а не строку.) Для той и другой формы (а мы рекомендуем первую) компилятор подсчитывает символы и таким образом получает размер массива.

Как и для других массивов, имя m1 является указателем на первый элемент массива:

m1 == &m1[0], \*m1 == 'Т', и \*(m1 + 1) == m1[1] == 'о'

Действительно, мы можем использовать указатель для создания строки. Например:

char \*m3 = " \n Достаточно обо мне — как вас зовут?";

Это почти то же самое, что и

static char m3[] = " \n Достаточно обо мне — как вас зовут?";

Оба описания говорят об одном: mЗ является указателем строки со словами " Как вас зовут?". В том и другом случае сама строка определяет размер памяти, необходимой для ее размещения. Однако вид их не идентичен.

### Различия: массив и указатель.

В чем же разница между этими двумя описаниями?

Описание с массивом вызывает создание в статической памяти массива из 38 элементов (по одному на каждый символ плюс один на завершающий символ ' \0'. Каждый элемент инициализируется соответствующим символом. В дальнейшем компилятор будет рассматривать имя mЗ как синоним адреса первого элемента массива, т. е. &m3[0]. Следует отметить, что mЗ является константой указателя. Вы не можете изменить m3, так как это означало бы изменение положения (адрес) массива в памяти. Можно использовать операции, подобные mЗ + 1, для идентификации следующего элемента массива, однако не разрешается выражение ++m3. Оператор увеличения можно использовать с именами переменных, но не констант.

Форма с указателем также вызывает создание в статической памяти 38 элементов для запоминания строки. Но, кроме того, выделяется еще одна ячейка памяти для переменной m3, являющейся указателем. Сначала эта переменная указывает на начало строки, но ее значение может изменяться. Поэтому мы можем использовать операцию увеличения; ++m3 будет указывать на второй символ строки (Д). Заметим, что мы не объявили \*m3 статической переменной, потому что мы инициализировали не массив из 38 элементов, а одну переменную типа указатель. Не существует ограничений на класс памяти при инициализации обычных переменных, не являющихся массивом.

Существенны ли эти отличия? Чаще всего нет, но все зависит от того, что вы пытаетесь делать.

Посмотрите несколько примеров, а мы возвращаемся к вопросу выделения памяти для строк.

**Массив и указатель: различия**

В нижеследующем тексте мы обсудим различия в использовании описаний этих двух видов

static char heart[] = "Я люблю Тилли";

char \*head = " Я люблю Милли'",

Основное отличие состоит в том, что указатель heart является константой, в то время как указатель head — переменной. Посмотрим, что на самом деле дает эта разница.

Во-первых, и в том и в другом случае можно использовать операцию сложения с указателем

for (i = 0, i < 6, i++ )

putchar(\*(heart + i) );

putchar( \n');

for (i = 0, i < 6, i++)

putchar(\*(head + i) );

putchar('\n'),

результате получаем

Я люблю

Я люблю

Но только в случае с указателем можно использовать операцию увеличения:

while ( \*(head) != ' \0' ) /\* останов в конце строки \*/

putchar( \*(head++ ) ); /\* печать символа и перемещение указателя \*/

дают в результате

Я люблю Милли!

Предположим, мы хотим заменить head на heart. Мы можем сказать

head = heart /\* теперь head указывает на массив heart \*/

но теперь мы можем сказать

heart = head, /\* запрещенная конструкция \*/

Ситуация аналогична х = 3 или 3 = х, левая часть оператора присваивания должна быть именем переменной. В данном случае head = heart, не уничтожит строку Милли, а только изменит адрес, записанный в head. Вот каким путем можно изменить обращение к heart и проникнуть в сам массив

heart [8] = 'М';

или

\*(heart + 8) = 'М';

Элементы массива (но не имя) являются переменными

### Явное задание размера памяти.

Иной путь выделения памяти заключается в явном ее задании. Во внешнем описании мы могли бы сказать:

char m1[44] = "Только ограничьтесь одной строкой.";

вместо

char m1[] = "Только ограничьтесь одной строкой." ;

Можно быть уверенным, что число элементов, по крайней мере, на один (это снова нуль-символ) больше, чем длина строки. Как и в других статических или внешних массивах, любые неиспользованные элементы автоматически инициализируются нулем (который в символьном виде является нуль-символом, а не символом цифры нуль).

**14_01**

Отметим, что в нашей программе массиву name задан размер:

char name[81];

Поскольку массив name должен читаться во время работы программы, у компилятора нет другого способа узнать заранее, сколько памяти нужно выделить для массива. Это не символьная константа, в которой компилятор может посчитать символы. Поэтому мы предположили, что 80 символов будет достаточно, чтобы поместить в массив фамилию пользователя.

### Массивы символьных строк.

Обычно бывает удобно иметь массив символьных строк. В этом случае можно использовать индекс для доступа к нескольким разным строкам. Покажем это на примере:

static char \*myta1[LIM] = {" Быстро складываю числа",

" Точно умножаю",

" Записываю данные" ,

" Правильно выполняю команды",

" Понимаю язык Си" };

Разберемся в этом описании. Вспомним, что LIM имеет значение 5, мы можем сказать, что mytal является массивом, состоящим из пяти указателей на символьные строки. Каждая строка символов, конечно же, представляет собой символьный массив, поэтому у нас есть пять указателей на массивы. Первым указателем является mytal [0], и он ссылается на первую строку. Второй указатель mytal[1] ссылается на вторую строку. Каждый указатель, в частности, ссылается на первый символ своей строки:

\*myta1[0] == 'Б', \*myta1[1] == 'Т', myta1[2] == 'З'

и т. д.

Инициализация выполняется по правилам, определенным для массивов. Тексты в кавычках эквивалентны скобочной записи

{{...}, {...}, ..., {...}};

где многоточия подразумевают тексты, которые мы поленились напечатать. В первую очередь мы хотим отметить, что первая последовательность, заключенная в двойные кавычки, соответствует первым парным скобкам и используется для инициализации первого указателя символьной строки. Следующая последовательность в двойных кавычках инициализирует второй указатель и т. д. Запятая разделяет соседние последовательности.

Кроме того, мы могли бы явно задавать размер строк символов, используя описание, подобное такому:

static char myta1[LIM][LINLIM];

Разница заключается в том, что второй индекс задает «прямоугольный» массив, в котором все «ряды» (строки) имеют одинаковую длину. Описание

static char \*myta1[LIM]

однако, определяет «рваный» массив, где длина каждого «ряда» определяется той строкой, которая этот «ряд» инициализировала. Рваный массив не тратит память напрасно.

**14_02**

### Указатели и строки.

Возможно, вы заметили периодическое упоминание указателей в нашем рассказе о строках. Большинство операций языка Си, имеющих дело со строками, работает с указателями. Например, рассмотрим приведенную ниже бесполезную, но поучительную программу

/\* указатели и строки \*/

#include<stdio.h>

#define PX(X) printf("X = %s; значение = %u; &X = %u\n" , X, X, &Х)

void main ()

{

static char \*mesg = " He делай глупостей!";

static char \*copy;

copy = mesg;

printf("%s\n" , copy);

PX(mesg);

PX(copy);

}

Взглянув на эту программу, вы можете подумать, что она копирует строку «Не делай глупостей!», и при беглом взгляде на вывод вам может показаться правильным это предположение:

Не делай глупостей!

mesg = Не делай глупостей!, значение = 14, &mesg = 32

сору = Не делай глупостей!, значение = 14, &сору = 34

Но изучим вывод РХ(). Сначала X, который последовательно является mesg и сору, печатается как строка (%s). Здесь нет сюрприза. Все строки содержат «Не делай глупостей!».

Далее ... вернемся к этому несколько позднее.

Третьим элементом в каждой строке является &Х, т. е. адрес X. Указатели mesg и сору записаны в ячейках 32 и 34 соответственно.

Теперь о втором элементе, который мы называем значением. Это сам X. Значением указателя является адрес, который он содержит. Мы видим, что mesg ссылается на ячейку 14, и поэтому выполняется сору.

Смысл заключается в том, что сама строка никогда не копируется. Оператор сору = mesg; создает второй указатель, ссылающийся на ту же самую строку.

Зачем все эти предосторожности? Почему бы не скопировать всю строку? Хорошо, а что эффективнее — копировать один адрес или, скажем, 50 отдельных элементов? Часто бывает, что адрес — это все, что необходимо для выполнения работы.

Теперь, когда мы обсудили определение строк в программе, давайте займемся вводом строк.

## Ввод и вывод строк.

Во многих приложениях более естественно выполнять ввод и вывод информации не посимвольно, а большими последовательностями. Например, файл торговых работников может иметь по одной записи в строке, а одна запись может содержать четыре поля: имя торгового работника, основная зарплата, комиссионные и количество проданного товара; при этом поля разделяются пробелами. В этом случае очень неудобно использовать символьный ввод/вывод.

Процесс ввода строки выполняется за два шага: выделение памяти для запоминания строки и применение функции ввода для получения строки.

### Выделение памяти.

Сначала следует определить место для размещения строки при вводе. Как было отмечено раньше, это значит, выделить память, достаточную для размещения любых строк, которые мы предполагаем читать. Не следует надеяться, что компьютер подсчитает длину строки при ее вводе, а затем выделит для нее память. Он не будет этого делать (если только вы не напишите программу, которая должна это выполнять). Если вы попытаетесь сделать что-то подобное

static char \*name;

scanf(" %s", name);

компилятор, вероятно, выполнит нужные действия. Но при вводе имя будет записываться на данные или текст вашей программы. Большинство программистов считает это очень забавным, но только в чужих программах.

Проще всего включить в описание явный размер массива:

char name[81];

Можно также использовать библиотечные функции языка Си, которые распределяют память, и мы рассмотрим их позднее.

В нашей программе для name использовался автоматический массив. Мы смогли это сделать, потому что не требовалось инициализации массива.

Как только выделена память для массива, можно считывать строку. Мы уже упоминали, что программы ввода не являются частью языка. Однако большинство систем имеют две библиотечные функции scanf() и gets(), которые могут считывать строки. Чаще всего используется функция gets(), поэтому мы вначале расскажем о ней.

### Использование функций gets(), puts(), fgets() и fputs().

**Функция gets()** имеет следующий прототип:

char \*gets(char \*s);

Она выполняет считывание строки символов из стандартного входного потока и помешает их в переменную s. Символ перехода на новую строку '\n' заменяется символом '\0' при помещении в строку s. При использовании этой функции следует соблюдать осторожность, чтобы число символов, считанных из входного потока, не превысило размер памяти, отведенной для строки s.

**Функция puts()** имеет следующий прототип:

int puts(const char \*s);

Она выводит строку в стандартный выводной поток и добавляет символ перевода на новую строку '\n'. В случае успеха puts () возвращает неотрицательное значение. В противном случае она возвращает EOF.

Приведем пример использования функции вывода строк:

#include <stdio.h>

using namespace std;

int main ()

{

setlocale(LC\_ALL, "Rus");

char\* string = "Выводимая строка";

puts(string);

getchar(); getchar();

return 0;

}

****

**Функция fgets()** имеет три аргумента, в то время как gets() имеет один.

Хороший способ чтения из файла дает функция fgets() (от "get string"):

char \*fgets(char \*buffer, size\_t length, FILE \*file);

Тут

* buffer -- это указатель на буфер, в который мы читаем;
* length -- это размер буфера;
* file -- это файл, из которого мы читаем (если читаем с клавиатуры, то разумно использовать stdin).

Функция возвращает строку. Эта функция делает примерно следующее. Она читает из файла file в буфер buffer не больше length-1 символов. Функция может прочитать не все length-1 символов в том случае, если она встретит конец строки, либо конец файла. Функция читает length-1 символ потому, что последний символ функция добавляет сама -- '\0'

Налицо быстрота и безопасность. Главное отличие от scanf-а заключается в том, что функция перестанет читать в тот момент, когда закончится буфер. Быстрота обусловлена тем, что функция scanf должна в момент выполнения разобрать форматную строку, в то время как fgets просто читает строку.

Вот пример ее использования:

/\* считывает файл строка за строкой \*/

#include <stdio.h>

#define MAXLIN 80

void main()

{

FILE \*f1;

char string[MAXLIN];

f1 = fopen("story.txt", "r");

while ( fgets(string, MAXLIN, f1) != NULL)

puts(string);

}

Первый из трех аргументов функции fgets() является указателем на местоположение считываемой строки. Мы располагаем вводимую информацию в символьный массив string.

Второй аргумент содержит +. Функция прекращает работу после считывания символа новой строки или после считывания символов общим числом MAXLIN — 1 (в зависимости от того, что произойдет раньше). В любом случае нуль-символ (' \0') добавляется в самый конец строки.

Третий аргумент указывает, конечно, на файл, который будет читаться.

Разница между gets() и fgets() заключается в том, что gets() заменяет символ новой строки на ' \0', в то время как fgets() сохраняет символ новой строки.

Подобно gets() функция fgets() возвращает значение NULL, если встречает символ EOF. Это позволяет вам проверить, как мы и сделали, достигли ли вы конца файла.

**Функция fputs()** очень похожа на puts(). Оператор

fputs(\* Вы были правы.", fileptr);

передает строку " Вы были правы." в файл, на который ссылается указатель fileptr типа FILE. Конечно, сначала нужно открыть файл при помощи функции fopen(). В самом общем виде это выглядит так

status = fputs(указатель строки, указатель файла);

где status является целым числом, которое устанавливается в EOF, если fputs() встречает EOF или ошибку.

Подобно puts() эта функция не ставит завершающий символ '\0' в конец копируемой строки. В отличие от puts() функция fputs() не добавляет символ новой строки в ее вывод.

Есть еще одно средство, которое может оказаться полезным, и мы его сейчас обсудим.

В соответствии со структурой файла лучше рассматривать каждую запись как символьную строку и единицу информации при считывании и записи. Для этого пригодна функция fgets(), которая считывает не отдельные символы, а целые строки. Вдобавок к функции fgets() и ее дополнению fputs() имеются аналогичные макросы gets() и puts().

Функция fgets() имеет три параметра: адрес массива, в котором хранится строка символов, максимальное число хранимых символов и указатель на считываемый файл. Эта функция считывает символы в массив до тех пор, пока количество введенных символов не будет на единицу меньше указанного размера (все символы, включая символ перевода строки) или не будет обнаружен признак конца файла.

Если fgets() считывает символ перевода строки, то он запоминается в массиве. Если считывается хотя бы один символ, то функция автоматически добавляет признак конца строки — null-символ \0. Предположим, что файл BOATSALE.DAT выглядит следующим образом:

Pat Pharr 32767 0.15 30 Beth Mollen 35000 0.12 23 Gary Kohut 40000 0.15 40

Если предположить, что максимальная длина строки равна 40 символам включая символ перевода строки, то следующая программа будет читать записи из файла и записывать их на стандартное устройство вывода:

/\*Программа на С, демонстрирующая считывание записей при помощи

функции fgets и их вывод на stdout при помощи функции fputs\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming\Universal\_HederFile.h"

void StopWait(void);

#define INULL\_CHAR 1

#define IMAX\_REC\_SIZE 40

main()

{

FILE \*pfinfile;

char crecord[IMAX\_REC\_SIZE + INULL\_CHAR];

pfinfile=fopen("E:\\LECTURE\\AlgorithmProgramming\\boatsale.dat", "r");

while(fgets(crecord,IMAX\_REC\_SIZE +INULL\_CHAR,pfinfile) != NULL)

fputs(crecord,stdout);

fclose(pfinfile) ;

StopWait(); /\* Wait a little \*/

return (0);

}

Поскольку максимальный размер записи равен 40, то необходимо зарезервировать 41 ячейку в памяти; дополнительная ячейка должна хранить null-символ \0. Программа не использует при печати записей дополнительный символ перевода строки; расчет делается на символ перевода строки, считанный в символьный массив функцией fgets(). Функция fputs() печатает содержимое символьного массива crecord в файл stdout, на который указывает указатель файла.

Если программа обращается к файлу в каталоге, отличном от того, в котором располагается компилятор, то в имени файла нужно указать полный путь. Обратите на это внимание в приведенном примере; две обратные наклонные черты (\\) синтаксически необходимы для указания подкаталога. Напоминаем, что одна наклонная черта (\) обычно указывает на то, что далее следует управляющая последовательность или продолжение строки.

Хотя функции gets() и fgets() используются аналогичным образом, функции puts() и fputs() работают по-разному. Функция fputs() записывает в файл и имеет два параметра: адрес символьной строки, заканчивающейся null-символом, и указатель на файл; fputs() просто копирует строку в указанный файл. Символ перевода строки в конце строки не добавляется.

Однако макрос puts() не требует указателя на файл, поскольку вывод автоматически назначается на stdout; в конце выводимой строки автоматически добавляется символ перевода строки.

Функция считывания строки gets() очень удобна для диалоговых систем. Она получает строку от стандартного устройства ввода вашей системы, которым, как мы предполагаем, является клавиатура. Поскольку строка не имеет заранее заданной длины, функция gets() должна знать, когда ей прекратить работу. Функция читает символы до тех пор, пока ей не встретится символ новой строки (' \n'), который вы создаете, нажимая клавишу [ввод]. Функция берет все символы до (но не включая) символа новой строки, присоединяет к ним нуль-символ ('\0') и передает строку вызывающей программе. Вот простой способ использования функции:

/\* получение имени1 \*/

#include <stdio.h>

void main ()

{

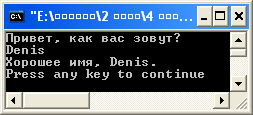
char name[81]; /\* выделение памяти \*/

printf("Privet, kak vas zovyt?\n");

gets(name); /\* размещение введенного имени в строку "name" \*/

printf("Horoshee imya, %s.\n", name);

}

****

Функция примет любое имя (включая пробелы) длиной до 80 символов. (Не забудьте запасти один символ для ' \0' .)

Отметим, что мы хотели при помощи функции gets() воздействовать на нечто (name) в вызывающей программе. Значит, нужно использовать указатель в качестве аргумента; а имя массива, конечно, является его указателем.

Функция gets () обладает большими возможностями, чем показано в последнем примере. Взгляните на эту программу:

/\* получение имени2 \*/

#include <stdio.h>

void main ()

{

char name [80];

char \*ptr, \*gets (char \*string);

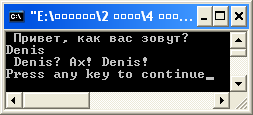
printf(" Привет, как вас зовут?\n");

ptr = gets(name);

printf(" %s? Ax! %s!\n", name, ptr);

}

Получился диалог:

****

Функция gets () предоставляет вам два способа ввода строки!

* Использует метод указателей для передачи строки в name.
* Использует ключевое слово return для возврата строки в ptr. Напомним, что ptr является указателем на тип char. Это означает, что gets() должна вернуть значение, которое является указателем на тип char. И в приведенном выше изложении вы можете увидеть, что мы так и описали gets ().

Описание вида

char \*gets ();

говорит о том, что gets() является функцией (отсюда круглые скобки) типа «указатель на тип char» (поэтому \* и char). В примере получение имени1 мы обходились без этого описания, потому что мы никогда не пытались использовать возвращенное значение функции gets().

Между прочим, вы можете также описать указатель на функцию. Это выглядело бы следующим образом:

char (\*foop) ();

и foop был бы указателем на функцию типа char. Мы расскажем немного подробнее о таких причудливых описаниях позднее.

Структура функции gets() выглядела бы примерно так:

char \*gets(s);

char \*s;

{

char \*p;

...

return (p);

}

На самом деле структура немного сложнее, и для gets () есть две возможности возврата. Если все идет хорошо, она возвращает считанную строку, как мы уже сказали. Если что-то неправильно или если gets() встречает символ EOF, она возвращает NULL, или нулевой адрес. Таким образом gets () включает разряд проверки ошибки.

Поэтому данная функция удобна для использования в конструкциях, подобных

while (gets(name) != NULL)

где NULL определен в файле stdio.h как 0. При помощи указателя массиву name присваивается значение. Наличие возврата позволяет присваивать значение всей gets(name) и выполнять проверку на EOF. Этот двоякий подход более компактен, чем использование функции getchar(), которая имеет возврат без аргумента.

while ( (ch = getchar ()) ! = EOF)

Мы должны полагаться на библиотечные функции, которые могут немного изменяться от системы к системе. Функции puts() и printf() — две рабочие лошадки, используемые при выводе строк.

Это очень простая функция; у нее есть только один аргумент, являющийся указателем строки. Нижеследующий пример иллюстрирует некоторые из многих способов ее применения.

/\* простые выдачи \*/

#include <stdio.h>

#define DEF "Я строка #define."

void main ()

{

static char str1[] = "Массив инициализирован мной ." ;

static char \*str2 = "Указатель инициализирован мной.";

puts(" Я аргумент функции puts ().");

puts(DEF);

puts(str1);

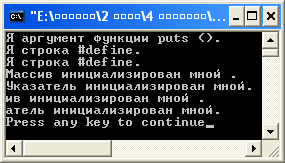
puts(str2);

puts(&str1[4]);

puts(str2+4);

}

В результате работы программы получаем

****

Этот пример напоминает нам, что фразы в кавычках и имена строк символьных массивов являются указателями. Обратите внимание на два последних оператора. Указатель &str1[4] ссылается на пятый элемент массива str1. Этот элемент содержит символ 'и', и функция puts () использует его в качестве начальной точки. Аналогично str2 + 4 ссылается на ячейку памяти, содержащую 'а' в «указателе», и с нее начинается вывод строки.

Как puts () узнает, когда остановиться? Она прекращает работу, если встречает нуль-символ, поэтому лучше, чтобы он был. Не пытайтесь делать так!

/\* нет строки1 \*/

main ()

{

static char dont[] = ('H', 'I' , '!','!');

puts(dont); /\* dont не является строкой \*/

}

Поскольку в dont отсутствует завершающий нуль-символ, она не является строкой. Так как нуль-символ отсутствует, puts () не знает, когда ей останавливаться. Она будет просто перебирать ячейки памяти, следующие за dont до тех пор, пока не найдет где-нибудь нуль-символ. Если повезет, она, может быть, найдет его в ближайшей ячейке, но может и не повезти.

Обратите внимание, что любая строка, вводимая функцией puts(), начинается с новой строки. Если puts() в конце концов находит завершающий нуль-символ, она заменяет его символом «новой строки» и затем выводит строку.

## Функции, работающие со строками.

Большинство библиотек языка Си снабжено функциями, работающими со строками. Рассмотрим четыре наиболее полезных и распространенных: strlen (), strcat (), strcmp () и strcpy().

Мы уже применяли функцию strlen (), которая находит длину строки. Используем ее в нижеследующем примере функции, укорачивающей длинные строки.

### Функция strlen().

/\* Функция Прокруста \*/

fit(string, size)

char \*string;

int size;

{

if (strlen(string) > size)

\*(string + size) = ' \0';

}

Проверьте ее в «деле» в этой тестовой программе:

/\* тест \*/

main ()

{

static char mesg[] = "Ну, теперь держитесь, компьютероманы.";

puts(mesg);

fit(mesg, 10);

puts(mesg);

}

Текст программы имеет вид:

#include<stdio.h>

#include <string.h>

void fit(char \*string, int size);

void main ()

{

static char mesg[] = "Ну, теперь держитесь, компьютероманы.";

puts(mesg);

fit(mesg, 10);

puts(mesg);

}

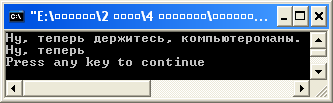
void fit(char \*string,int size)

{

if (strlen(string) > size)

\*(string + size) = ' \0';

}

****

Наша функция помешает символ '\0' в одиннадцатый элемент массива, заменяя символ пробела. Остаток массива остается на старом месте, но puts () прекращает работу на первом нуль-символе и игнорирует остаток массива.

### Функция strcat().

Вот что умеет делать функция strcat ():

/\* объединение двух строк \*/

#include <stdio.h>

#include<string.h>

void main ()

{

static char flower [80];

static char addon[] = "ы пахнут старыми ботинками.";

puts(" Назовите ваш любимый цветок.");

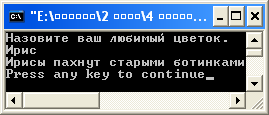
gets(flower);

strcat(flower, addon);

puts(flower);

}

Получаем на экране:

****

Очевидно, что strcat () (string concatenation) использует в качестве аргументов две строки. Копия второй строки присоединяется к концу первой, и это объединение становится новой первой строкой. Вторая строка не изменяется.

Внимание! Эта функция не проверяет, умещается ли вторая строка в первом массиве. Если вы ошиблись при выделении памяти для первого массива, то у вас возникнут проблемы. Конечно, можно использовать strlen () для определения размера строки до объединения.

/\* Объединение двух строк, проверка размера первой \*/

#include< stdio.h>

#include<string.h>

#define SIZE 80

void main ()

{

static char flower [SIZE];

static char addon[] = "ы пахнут старыми ботинками.";

puts("Назовите ваш любимый цветок.");

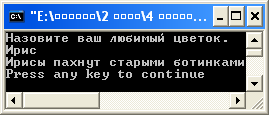
gets(flower);

if ( (strlen(addon) + strlen(flower) + 1) < SIZE)

strcat(flower, addon);

puts(flower);

}

****

Мы добавляем 1 к объединенной длине для размещения нуль-символа.

### Функция strcmp().

Предположим, что вы хотите сравнить чей-то ответ со строкой, находящейся в памяти:

/\* Будет ли это работать? \*/

#include <stdio.h>

#include <string.h>

#define ANSWER "Grant"

main ()

{

char chance [40];

puts("Who was buried in the Grant's grave?");

gets(chance);

while(chance != ANSWER)

{

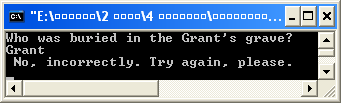
puts(" No, incorrectly. Try again, please.");

gets(chance);

}

puts (" You all right!");

}

****

Хотя эта программа и смотрится неплохо, она не будет работать правильно,chance и ANSWER на самом деле являются указателями, поэтому сравнение (chance != ANSWER) спрашивает не о том, одинаковы ли эти две строки, а одинаковы ли два адреса, на которые ссылаются chance и ANSWER. Так как ANSWER и chance запоминаются в разных ячейках, эти два указателя никогда не могут быть одним и тем же, и пользователю всегда сообщается, что программа неверна. Такие программы обескураживают людей.

Нам нужна функция, которая сравнивает содержимое строк, а не их адреса. Можно было бы придумать ее, но это уже сделала за нас функция strcmp () (string comparision).

Теперь исправим нашу программу:

/\* это будет работать \*/

#include <stdio.h>

#include <string.h>

#define ANSWER "Grant"

void main ()

{

char chance [40];

puts("Who was buried in the Grant's grave?");

gets(chance);

while (strcmp(chance, ANSWER) != 0)

{

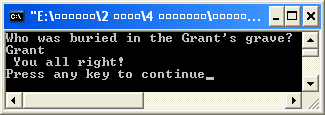
puts(" No, incorrectly. Try again, please.");

gets(chance);

}

puts(" You all right!");

}

****

Так как ненулевые значения интерпретируются всегда как «true», мы можем сократить оператор while до while (strcmp(chance, ANSWER)).

Из этого примера можно сделать вывод, что strcmp () использует два указателя строк в качестве аргументов и возвращает значение 0, если эти две строки одинаковы. Прекрасно, если вы придете к такому выводу.

Хорошо, что strcmp() сравнивает строки, а не массивы. Поэтому, хотя массив chance занимает 40 ячеек памяти, а " Grant" — только 6 (не забывайте, что одна нужна для нуль-символа), сравнение выполняется только с частью try, до его первого нуль-символа. Такую функцию Strcmp () можно использовать для сравнения строк, находящихся в массивах разной длины.

А что если пользователь ответил " GRANT" или "grant" или "Улиссес С. Грант"? Хорошо, если пользователю сказали, что он ошибся? Чтобы сделать программу гибкой, вы должны предусмотреть несколько допустимых правильных ответов. Здесь есть некоторые тонкости. Вы могли бы в операторе #define определить в качестве ответа "GRANT" и написать функцию, которая превращает любой ответ только в это слово. Это устраняет проблему накопления, но остаются другие поводы для беспокойства.

Между прочим, какое значение возвращает strcmp(), если строки не одинаковы? Вот пример:

/\* возвраты функции strcmp \*/

#include <stdio.h>

#include<string.h>

void main ()

{

printf("%d\n" , strcmp("A" , "A"));

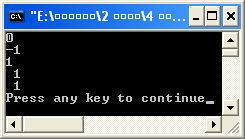
printf("%d\n" , strcmp("A" , "B"));

printf("%d\n" , strcmp("B" , "A"));

printf(" %d\n", strcmp("C" , "A"));

printf(" %d\n" , strcmp("apples" , "apple"));

}

****

В результате получаем

0

-1

1

2

115

Как мы и предполагали, сравнение "А" с самим собой возвращает 0. Сравнение "А" с "В" дает -1, а "В" с "А" дает 1. Это наводит на мысль, что strcmp() возвращает отрицательное число, если первая строка предшествует второй в алфавитном порядке, или положительное число, если порядок иной. Кроме того, сравнение "С" с "А" дает 2 вместо 1. Картина проясняется: функция возвращает разницу между двумя символами в коде ASCII. В более общем смысле strcmp() передвигается вдоль строк до тех пор, пока не находит первую пару несовпадающих символов; затем она возвращает разницу в кодах ASCII. Например, в самом последнем примере "apples" и "apple" совпадают, кроме последнего символа 's', в первой строке. Он сопоставляется с шестым символом в "apple", который является нуль-символом (0 в ASCII).

Возвращается значение

's' - '\0' = 115 - 0 = 115,

где 115 является кодом буквы 's' в ASCII.

Обычно вам не нужно точно знать возвращаемое значение. Чаще всего вы только хотите знать, нуль это или нет, т. е. было ли совпадение. Или, может быть, вы пытаетесь отсортировать строки в алфавитном порядке и хотите узнать, в каком случае сравнение дает положительный, отрицательный или нулевой результат.

Можно использовать эту функцию, чтобы проверить, остановится ли программа, читая вводимую информацию:

/\* Начало какой-то программы \*/

#include < stdio.h>

#define SIZE 81

#define LIM 100

#define STOP " "/\* нулевая строка \*/

main ()

{

static char input [LIM] [SIZE];

int ct = 0;

while(gets(input[ct]) != NUL && strcmp(input[ct], STOP) != 0 && ct++ < LIM)

...

}

Программа прекращает чтение вводимой строки, если встречает символ EOF [в этом случае gets () возвращает NULL], или если вы нажимаете клавишу [ввод] в начале строки (т. е. введете пустую строку), или если вы достигли предела LIM. Чтение пустой строки дает пользователю простой способ прекращения ввода.

Давайте перейдем к последней из обсуждаемых нами функций, работающих со строками.

### Функция strcpy().

Мы уже говорили, что если pts1 и pts2 являются указателями строк, то выражение

pts2 = pts1;

копирует только адрес строки, а не саму строку. Предположим, что вы все же хотите скопировать строку. В этом случае можно использовать функцию strcpy (). Она работает примерно так:

/\* демонстрация strcpy () \*/

#include <stdio.h>

#include <string.h>

#define WORDS "Проверьте, пожалуйста, вашу последнюю запись. "

void main ()

{

static char \*orig = WORDS;

static char copy [40];

puts(orig);

puts(copy);

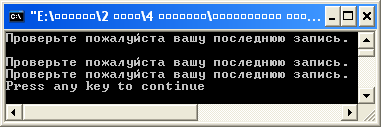
strcpy(copy, orig);

puts(orig);

puts(copy);

}

Вот результат:

****

Очевидно, что строка, на которую указывает второй аргумент (orig) функции strcpy (), скопирована в массив, на который указывает первый аргумент (сору). Порядок аргументов функции такой же, как в операторе присваивания: строка, получающая значение, стоит слева. (Пустая строка является результатом печати массива сору до копирования, и она говорит о том, что статические массивы инициализируются нулями, т. е. нуль-символами в символьном виде.)

Нужно обеспечить, чтобы размер массива, принимающего строку, был достаточен для ее размещения. Поэтому мы используем описание

static char copy [40];

а не

static char \*copy; /\* не выделяет память для строки \*/

Короче говоря, strcpy() требует два указателя строк в качестве аргументов. Второй указатель, ссылающийся на исходную строку, может быть объявленным указателем, именем массива или строковой константой. А первый указатель, ссылающийся на копию, должен ссылаться на массив или часть массива, имеющего размер, достаточный для размещения строки.

Теперь, когда мы описали несколько функций, работающих со строками, рассмотрим целую программу, работающую со строками.

**Пример: сортировка строк.**

Возьмем реальную задачу сортировки строк в алфавитном порядке. Эта задача может возникнуть при подготовке списка фамилий, при создании алфавитного указателя и во многих других ситуациях. В такой программе одним из главных инструментов является функция strcmp(), так как ее можно использовать для определения старшинства двух строк. Последовательность наших действий будет состоять из считывания массива строк, их сортировки и последующего вывода. Совсем недавно мы показали последовательность действий для считывания строк, и сейчас мы начнем программу таким же образом.

/\* считывает строки и сортирует их \*/

#include <stdio.h>

#define SIZE 81 /\* предельная длина строки, включая\0 \*/

#define LIM 20 /\* максимальное количество считываемых строк \*/

#define HALT " " /\* нулевая строка для прекращения ввода \*/

main ()

{

static char input [LIM] [SIZE]; /\* массив для запоминания вводимых строк \*/

char \*ptstr[LIM]; /\* массив переменных типа указатель \*/

int ct = 0; /\* счетчик вводимых строк \*/

int k; /\* счетчик выводимых строк \*/

printf(" Введите до %d строк и я их отсортирую.\n" , LIM);

printf(" Для прекращения ввода нажмите клавишу [ввод] в начале строки\n");

while( ( gets(input[ct]) != NULL) && strcmp(input[ct], HALT) != 0&& ct++ < LIM)

ptstr[ct - 1] = input [ct - 1]; /указывает на еще не отсортированный ввод \*/

stsrt(ptstr, ct); /\* сортировка строк \*/

puts(\* \nВот отсортированный список строк:\n");

for (k = 0; k < ct; k++ )

puts(ptstr[k]); /\* указатели на отсортированные строки \*/

}

/\* функция сортировки строк с использованием указателей \*/

stsrt(strings, num)

char \*strings[];

int num;

{

char \*temp;

int top, seek;

for (top = 0; top < num - 1; top++ )

for (seek = top + 1; seek < num; seek++)

if (strcmp(strings[top], strings[seek]) > 0)

{

temp = strings [top];

strings [top] = strings [seek];

strings [seek] = temp;

}

}

Исходный текст программы будет иметь вид:

/\* считывает строки и сортирует их \*/

#include <stdio.h>

#include <string.h>

#define SIZE 81 /\* предельная длина строки, включая\0 \*/

#define LIM 20 /\* максимальное количество считываемых строк \*/

#define HALT " " /\* нулевая строка для прекращения ввода \*/

void stsrt(char \*strings[], int num)

{

char \*temp;

int top, seek;

for (top = 0; top < num - 1; top++ )

for (seek = top + 1; seek < num; seek++)

if (strcmp(strings[top], strings[seek]) > 0)

{

temp = strings [top];

strings [top] = strings [seek];

strings [seek] = temp;

}

}

void main ()

{

static char input [LIM] [SIZE]; /\* массив для запоминания вводимых строк \*/

char \*ptstr[LIM]; /\* массив переменных типа указатель \*/

int ct = 0; /\* счетчик вводимых строк \*/

int k; /\* счетчик выводимых строк \*/

printf(" Введите до %d строк и я их отсортирую.\n" , LIM);

printf(" Для прекращения ввода нажмите клавишу [ввод] в начале строки\n");

while( ( gets(input[ct]) != NULL) && strcmp(input[ct], HALT) != 0&& ct++ < LIM)

ptstr[ct - 1] = input [ct - 1]; /\*указывает на еще не отсортированный ввод \*/

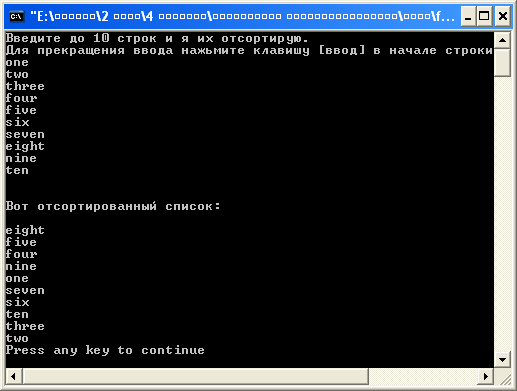
stsrt(ptstr, ct); /\* сортировка строк \*/

puts(" \nВот отсортированный список строк:\n");

for (k = 0; k < ct; k++ )

puts(ptstr[k]); /\* указатели на отсортированные строки \*/

}

****

Вывод строк на печать не составляет проблемы, а для сортировки можно взять тот же алгоритм, который использовался раньше для чисел. Сейчас мы применим один хитрый трюк: посмотрим, сможете ли вы его заметить.

Для проверки возьмем детский стишок.

Введите до %d строк и я их отсортирую.

Для прекращения ввода нажмите клавишу [ввод] в начале строки

Жил на свете человек

Скрюченные ножки

И гулял он целый век

По скрюченной дорожке

Вот отсортированный список строк

Жил на свете человек

И гулял он целый век

По скрюченной дорожке

Скрюченные ножки

Детские стишки не кажутся слишком искаженными после сортировки их по алфавиту.

Трюк состоит в том что вместо перегруппировки самих строк мы перегруппировали их указатели. Разберемся в этом. В начале ptrst[0] ссылается на input[0] и т. д. Каждый input[] является массивом из 81 элемента, а каждый элемент ptrst[] является отдельной переменной. Процедура сортировки перегруппировывает ptrst, не трогая input. Если, например, input[1] стоит перед input[0] по алфавиту, то программа переключает указатели ptrst, в результате чего ptrst[0] ссылается на input[1], a ptrst[1] на input[0]. Это гораздо легче, чем, используя strcpy (), менять местами две введенные строки. Просмотрите еще раз этот процесс на рисунке.

И наконец, давайте попытаемся заполнить пробелы, оставшиеся в нашем описании, а именно «пустоту» между скобками в функции main().

## Создание собственных функций ввода/вывода.

Не ограничивайте себя при вводе и выводе только этими библиотечными функциями. Если у вас нет нужной функции, или она вам не нравится, можно создавать свои собственные версии, используя для этого getchar() и putchar().

Предположим, у вас нет функции puts(). Вот один из путей ее создания:

/\* put1---- печатает строку \*/

put1 (string);

char \*string;

{

while(\*string != ' \0')

putchar(\*string++ );

putchar(' \n');

}

Символьный указатель string вначале ссылается на первый элемент вызванного аргумента. После печати его содержимого указатель увеличивается и ссылается уже на следующий элемент. Это продолжается до тех пор, пока указатель не дойдет до элемента, содержащего нуль-символ. Затем в конце строки будет поставлен символ новой строки.

Предположим, у вас есть puts(), но вам нужна функция, которая, кроме того, сообщает, сколько напечатано символов. Эту возможность легко добавить:

/\* put2-----печатает строку и считывает символы \*/

put2 (string);

char \*string;

{

int count = 0;

while (\*string != ' \0')

{

putchar(\*string ++);

count ++;

}

putchar(' \n');

return(count);

}

Вызов

put2("пицца");

печатает строку пицца, в то время как оператор

num = puts("пицца");

передаст, кроме того, количество символов в num; в данном случае это число 5. Вот несколько более сложный вариант, показывающий вложенные функции:

/\* вложенные функции \*/

#include <stdio.h>

main ()

{

put1 ("Если бы я имел столько денег, сколько могу потратить,");

рrintf("Я считаю %d символа\n",

put2 ("Я никогда бы не жаловался, что приходится чинить старые стулья."));

}

Исходный текст программы будет иметь следующий вид:

/\* вложенные функции \*/

#include <stdio.h>

/\* put1---- печатает строку \*/

void put1 (char \*string)

{

while(\*string != '\0')

putchar(\*string++ );

putchar(' \n');

}

/\* put2-----печатает строку и считывает символы \*/

int put2 (char \*string)

{

int count = 0;

while (\*string != '\0')

{

putchar(\*string ++);

count ++;

}

putchar(' \n');

return(count);

}

void main ()

{

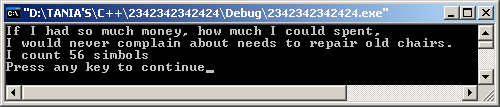
put1 ("If I had so much money, how much I could spent,");

printf("I count %d simbols\n", put2 ("I would never complain about needs to repair old chairs."));

}

(Мы включили в программу при помощи директивы #include файл Stdio.h, потому что в нашей системе в нем определена функция putchar(), а она используется в нашей новой функции.)

Да-а, мы используем функцию printf() для печати значения put2(), но в процессе нахождения значения put2() компьютер должен сначала заставить ее поработать — напечатать строку. Вот что получается при этом:

****

Теперь вы можете построить работающую версию функции gets (); она должна быть похожа на нашу функцию getint (), но гораздо проще ее.

## Проверка и преобразование символов.

Заголовочный файл ctype.h содержит несколько функций макроопределений, которые проверяют, к какому классу принадлежат символы. Функция isalpha(c), например, возвращает ненулевое значение (истина), если с является символом буквы, и нуль (ложь), если символ не является буквой. Таким образом,

isalpha('S') != 0, но isalpha('#') ==0

Ниже перечислены функции, чаще всего находящиеся в этом файле. В каждом случае функция возвращает ненулевое значение, если с принадлежит к опрашиваемому классу, и нуль в противном случае.

**ФУНКЦИЯ ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ c**

isalpha(c) буквой

isdigit(c) цифрой

islower(c) строчной буквой

isspace(c) пустым символом (пробел, табуляция или новая строка)

isupper(c) прописной буквой

Ваша система может иметь дополнительные функции, такие как

**ФУНКЦИЯ ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ c**

isalnum(c) алфавитно-цифровым (буква или цифра)

isascii(c) кодом ASCII (0—127)

iscntrl(c) управляющим символом

ispunct(c) знаком пунктуации

Еще две функции выполняют преобразования:

toupper(c) преобразует с в прописную букву

tolower(c) преобразует с в строчную букву

В некоторых системах преобразование выполняется только в случае, если символ находится в регистре (прописных или строчных букв), противоположном тому, с которого следует начинать. Однако надежнее предварительно проверить регистр.

Ниже дана программа, использующая некоторые из этих функций для преобразования всего файла в прописные, или строчные буквы, по вашему желанию. Для получения небольшого разнообразия используем диалоговый подход, вместо того чтобы применять аргументы командной строки для снабжения программы информацией.

/\* преобразование строчных букв в прописные и обратно \*/

#include <stdio.h>

#include < ctype.h> /\* включает файл макроопределений \*/

#define. UPPER 1

#define LOWER 0

main()

{

int crit; /\* для установки регистра прописных или строчных букв \*/

char file1[14], file2[14]; /\* имена входного и выходного файлов \*/

crit = choose(); /\* выбирает прописные или строчные буквы \*/

getfiles(file1, file2); /\* получает имена файлов \*/

conv(file1, file2, crit); /\* выполняет преобразование \*/

}

choose()

{

int ch;

printf(" Программа преобразует весь файл в прописные буквы или \n");

printf(" в строчные буквы. Вводит U, если нужны прописные буквы\n");

printf(" или вводит L, если нужны строчные буквы.\n");

while ( (ch = getchar()) != 'U' && ch != 'u' && ch ! = 'L' && ch !='!')

printf(" Введите, пожайлуста, U или L.\n");

while (getchar() != '\n')

; /\* сбрасывает последний символ новой строки \*/

if ( ch == 'U' || ch == 'u')

{

printf(" Все в порядке, есть регистр прописных букв.");

return(UPPER);

}

else

{

printf(" Все в порядке, есть регистр строчных букв." );

return(LOWER);

}

}

getfiles(name1, name2);

char \*name1, name2;

{

printf(" Какой файл вы хотите преобразовать? \n" );

gets(name1);

printf(" Это\" %s\" \n", name1);

printf(" Какое имя вы хотите выбрать для преобразуемого файла? \n");

while ( strcmp(gets(name2), name1) == NULL)

printf(" Выберите другое имя\n" );

printf(" Ваш выходной файл\" %s\" \n", name2);

}

conv(name1, name2, crit)

char \*name1, name2;

int crit;

{

int ch;

FILE \*f1, \*f2;

if ( (f1 = fopen(name1, "r")) == NULL)

printf("Извините, я не могу открыть %s. До свидания.\n" , name1);

else

{

puts(" Итак, начнем!");

f2 = fopen(name2, "w");

while ( (ch = getc(f1)) != EOF)

{

if (crit == UPPER)

ch = islower(ch) ? toupper(ch) : ch;

else

ch = isupper(ch) ? tolower(ch) : ch;

putc(ch, f2);

}

fclose(f2);

fclose(f1);

puts(" Сделано!");

}

}

Мы разделили программу на три части: получение от пользователя указания о виде преобразования, получение имени входного и выходного файлов и выполнение преобразования. Чтобы осуществить все это, мы создали разные функции для каждой -части.

Функция choose() довольно проста за исключением, может быть, цикла

while( getchar() ! = ' \n')

;

Этот цикл включен для решения проблемы, с которой мы столкнулись раньше. Когда пользователь отвечает на вопрос о виде преобразования, скажем, буквой U, он нажимает клавишу U, а затем клавишу [ввод], которая передает ' \n'. Первоначальная функция getchar() извлекает U, но оставляет ' \n' для следующего чтения строки. Функция gets(), входящая в getnames(), интерпретировала бы ' \n' как пустую строку, поэтому мы использовали малый цикл while, чтобы избавиться от символа «новая строка». Действительно, простая getchar(), сделала бы это, если бы пользователь непосредственно за U нажимал бы [ввод]. Но наша версия, кроме того, предусматривает возможность нажать на клавишу пробела несколько раз перед [ввод].

В функции getnames() для вас не должно быть сюрпризов. Учтите, что мы запрещаем пользователю применять одинаковые имена для выходного и входного файлов. Стандартная версия функции fopen() не позволяет вам и читать и записывать один и тот же файл, если вы открыли его один раз.

Функция conv() является функцией копирования с выполнением преобразования. Значение crit используется для определения требуемого преобразования. Работа выполняется простым условным оператором, таким как

ch = islower(ch) ? toupper(ch) : ch;

Он проверяет, является ли ch строчной буквой. Если да, то символ преобразуется в прописную букву. Если нет, остается как есть.

Макрофункции файла ctype.h предоставляют удобные и полезные средства для программирования. Теперь давайте займемся некоторыми более сложными функциями преобразования.

## Преобразования символьных строк: atoi(), atof().

Использование scanf() для считывания цифровых значений не является самым надежным способом, поскольку scanf() легко ввести в заблуждение ошибками пользователей при вводе чисел с клавиатуры. Некоторые программисты предпочитают считывать даже числовые данные как символьные строки и преобразовывать строку в соответствующее числовое значение. Для этого используются функции atoi() и atof(). Первая преобразует строку в целое, вторая — в число с плавающей точкой. Вот образец их использования:

/\* включение atoi() \*/

#include <stdio.h>

#include <windows.h>

#include<ctype.h>

#define issign(c) ( ((c) =='-' || (c) =='+')? (1) : (0))

#define SIZE 10

#define YES 1

#define NO 0

void main()

{

char ch;

static char number [SIZE];

int value;

int digit = YES;

int count = 0;

puts(" Enter integer number.");

gets(number);

if (number[SIZE - 1] != '\0')

{

puts(" Too many numbers; You killed me.");

exit(1);

}

while ( (ch = number [count]) != '0' && digit == YES)

if( !issign(ch) && !isdigit(ch) && !isspace(ch) )

digit = NO;

if ( digit == YES)

{

value = atoi(number);

printf("The number was %d.\n" , value);

}

else

printf(" It doesn't look like integer.");

}

Мы предусмотрели проверку некоторых ошибок.

Во-первых, следует посмотреть, умещается ли входная строка в предназначенном для нее массиве. Поскольку number является статическим символьным массивом, он инициализируется нулями. Если последний элемент массива не является нулем, значит что-то неверно, и программа прекращает работу. Здесь мы использовали библиотечную функцию exit(), которая выводит нас из программы. Немного позже мы расскажем кратко об этой функции.

Затем посмотрим, не содержит ли строка что-нибудь кроме пробелов, цифр и алгебраических знаков. Функция отвергает такие строки, как «дерево» или «1.2Е2». Ее устраивает смесь, подобная «3 - 4 + 2», но atoi() будет выполнять дальнейший отбор. Вспомним, что ! является операцией отрицания, поэтому !isdigit(c) означает: «с не является цифрой».

Строка

value = atoi(number);

показывает, как используется функция atoi(). Ее аргумент является указателем символьной строки; в этом случае мы применили имя массива number. Функция возвращает целое значение для такой строки. Таким образом, «1234» является строкой из четырех символов и переводится в 1234 — единое число типа int.

Функция atoi() игнорирует ведущие пробелы, обрабатывает ведущий алгебраический знак, если он есть, и обрабатывает цифры вплоть до первого символа, не являющегося цифрой. Поэтому наш пример «3 - 4 + 2» был бы превращен в значение 3.

Функция atof() выполняет подобные действия для чисел с плавающей точкой. Она возвращает тип double, поэтому должна быть описана как double в использующей ее программе.

Простые версии atof() будут обрабатывать числа вида 10.2, 46 и —124.26. Более мощные версии преобразуют также экспоненциальную запись, т. е. числа, подобные 1.25Е — 13.

Ваша система может также иметь обратные функции, работающие в противоположном направлении. Функция itоа() будет преобразовывать целое в символьную строку, а функция ftoa() преобразовывать число с плавающей точкой в символьную строку.

В то время как фунция fgets читеат обычную строку, функция scanf может читать и различные другие типы (целые, вещественные числа).

В языке C есть семейство функций - atoi (a -- ASCII, i -- integer):

N = atoi(string);

Функция принимает единственный параметр строку и пытается ее привести в типу int. Надо заметить, что функция atoi безопасная, но не очень удобная. Безопасная в том смысле, что не сломается: atoi("25a") == 25. "Неудобства" заключаются в том, то если мы передаем в качестве параметра строку, в которой есть не только числа, нужно быть очень внимательным и знать, как работает эта функция. Функция atoi никак не проинформирует нас, если преобразование прошло неудачно.

Например, atoi("abc") == 0, что на самом деле не совсем соответствует действительности. Использовать функцию atoi нужно лишь в том случае, когда вы уверены, что в строке есть число.

Родственные функции: atol, atoll, atof, strtol.

Им соответствуют функции для преобразования в типы long, long long и float.

Рассмотрим подробнее strtol:

long strtol(char \*buffer, char \*\*endPtr, int base);

Тут

buffer -- это указатель на буфер, из которого мы читаем;

endPtr -- это переменная, которая используется для того, чтобы сообщить нам насколько успешно произошло преобразование; это указатель на char \*, в котором записан первый символ, который не смог проинтерпретироваться с помощью функции strtol;

Применение выглядит примерно так:

char \*end;

char \*ptr = "25a";

int N = strtol(ptr, &end, 10);

Теперь end указывает на "a".

if (ptr == end) {

// ничего не получилось прочитать

}

* base -- это основание системы счисления, с которой мы работаем (от 2 до 36);
* Функция возвращает целое число типа long.

**Более мощное средство**

Есть более мощное средство, чем нежели fgets + atoi. Речь идет о функции sscanf. Вместо использования функции fscanf(f, "%d", &N) можно использовать связку:

fgets(ptr, 100, f);

sscanf(ptr, "%d", &N);

В чем преимущество и мощность такого подхода?

* Мы знаем длину того, что мы прочитали;
* Рассмотрим следующую ситуацию: мы хотим прочитать какие-то данные, но не смогли из-за ошибки.

Используя такой подход, мы можем передать пользователю сообщение об ошибке и ту строчку, которую нам не удалось прочитать: ptr. С использованием fscanf-а это невозможно.

С помощью fscanf-а мы можем только узнать, сколько переменных было успешно прочитан. А именно: посмотреть на возвращаемое значение.

При таком подходе, осталась невысокая скорость работы, однако надежность есть.

## Ввод и вывод целых чисел.

В некоторых программах возникает необходимость ввода и вывода в поток (или в буфер) целочисленной информации. Для этого в С имеется две функции: getw() и putw().

### Использование функций getw() и putw().

Дополняющие друг друга функции getw() и putw() весьма похожи на функции getc() и putc(); отличие в том, что они работают с целыми числами, а не с символами. Функции getw() и putw() можно использовать только с файлами, открытыми в двоичном режиме. Следующая программа открывает файл в двоичном режиме, записывает в него десять целых чисел, закрывает файл, а затем заново открывает файл на считывание и распечатывает числа:

/\*Программа на С, использующая функции getw и putw с файлом,

открытом в двоичном режиме\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming\Universal\_HederFile.h"

void StopWait(void);

#define ISIZE 10

int main()

{

FILE \*pfi;

int ivalue,ivalues[ISIZE],i;

pfi = fopen("E:\\LECTURE\\AlgorithmProgramming\\integer.dat", "wb");

if (pfi = NULL)

{

printf("File could not be opened"); /\* Файл невозможно открыть \*/

exit(1);

for(i = 0; i < ISIZE; i++)

{

ivalues [i]=i- NULL ;

putw (ivalues [i] ,pfi) ;

}

fclose (pfi) ;

pfi=fopen("E:\\LECTURE\\AlgorithmProgramming\\integer.dat", "r+b");

if(pfi = NULL)

{

printf("File could not be re-opened"); /\* Файл невозможно открыть \*/

/\* заново \*/

exit(1);

}

while (!feof (pfi))

{

ivalue = getw(pfi);

printf("%3d",ivalue) ;

}

}

StopWait(); /\* Wait a little \*/

return (0);

}

Посмотрите на результат работы программы и определите, что неправильно:

1 2 3 4 5 6 7 8 9 10 -1

Поскольку целое значение, считываемое в последнем цикле, может иметь значение EOF, для проверки признака конца файла в программе используется функция feof(). Однако, эта функция не выполняет опережающего считывания, как делается в подобных функциях в некоторых других языках высокого уровня. В С для определения этого условия необходимо реально выполнить операцию считывания значения "конец файла".

Для того чтобы исправить эту ситуацию, программу нужно переписать, используя так называемый оператор предварительного считывания:

/\*Программа на С, использующая функции getw и putw с файлом,

открытом в двоичном режиме\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming\Universal\_HederFile.h"

void StopWait(void);

#define ISIZE 10

main()

{

FILE \*pfi;

int ivalue,ivalues[ISIZE],i;

pfi = fopen("E:\\LECTURE\\AlgorithmProgramming\\integer.dat ", "w+");

if (pfi == NULL)

{

printf("File could not be opened"); /\* Файл невозможно открыть \*/

exit(1) ;

}

for(i = 0; i < ISIZE; i++)

{

ivalues[i]=i+1;

putw(ivalues[i],pfi);

}

fclose (pfi);

pfi=fopen("E:\\LECTURE\\AlgorithmProgramming\\integer.dat ", "r+");

if(pfi == NULL)

{

printf("File could not be re-opened"); /\* Файл невозможно открыть \*/

/\* заново \*/

exit(1);

}

while(!feof(pfi))

{

ivalue=getw(pfi);

printf("%3d",ivalue);

}

StopWait(); /\* Wait a little \*/

return (0);

}

Перед тем как в программе начинается последний цикл while, выполняется предварительное считывание для того, чтобы определить, не пустой ли файл, и, если это так, то действующее целое значение запоминается в переменной ivalue. Однако, если файл пустой, то функция feof() распознает это, и цикл while не выполняется.

Обратите также внимание на то, что предварительное считывание вызывает перестановку операторов внутри цикла while. При входе в цикл переменная ivalue содержит действующее целое значение. Если бы операторы в цикле оставались теми же, что и в предыдущей программе, то сразу же выполнился бы второй вызов функции getw() и при этом было бы потеряно первое целое значение. Из-за наличия предварительного считывания первым оператором в цикле while должен быть оператор вывода, за которым следует вызов функции getw() для считывания следующего значения.

Предположим, что цикл while выполнился девять раз. К концу девятого прохода выполнилась эхо-печать целых чисел от 1 до 8 и переменной ivalue присвоено значение 9. При следующем проходе цикла печатается 9 и вводится значение 10. Поскольку значения 10 и EOF не совпадают, цикл повторяется снова, в результате чего печатается значение 10 и считывается EOF. В этот момент цикл прекращается, поскольку функция feof() распознает условие конца файла.

Два приведенных примера иллюстрируют необходимость в соблюдении осторожности при написании программ с функцией feof(). Это — специфическая программистская проблема, так как каждый язык высокого уровня может трактовать условие конца файла по-своему. В некоторых языках при считывании данных выполняется предварительный анализ возможного на следующем этапе условия конца файла; в других языках, таких как С, этого не происходит.

## Форматированный вывод.

Богатый ассортимент средств управления форматом вывода в С позволяет легко создавать печатные графики, отчеты или таблицы. Двумя основными функциями, выполняющими этот форматированный вывод, являются printf() и эквивалентная функция для файлов — fprintf(). Эти функции могут использовать любые символы преобразования, показанные в табл. 11.2. Спецификация формата имеет следующий вид:

%[флаги] [ширина] [.точность] [{h | 1 | L}]тип

Таблица 11.2. Символы преобразования для функций printf() и fprintf()

|  |  |  |
| --- | --- | --- |
| **ПОЛЕ “ТИП”** | | |
| **Сим-**  **вол** | **Тип** | **Формат вывода** |
| c | int или wint\_t | printf—означает однобайтный символ  wprintf—означает широкий символ |
| C | int или wint\_t | printf—означает широкий символ  wprintf—означает однобайтный символ |
| d | int | Десятичное целое зо знаком |
| e | double | Число со знаком в виде [-]d.ddd e [знак]ddd,где d-отдельная десятичная цифра, ddd-одна или несколько десятичных цифр, ddd-ровно четыре десятичные цифры и знак “+” или ”-” |
| E | double | То же что и “е”, за исключением того, что перед экспонентой стоит “E” |
| f | double | Число со знаком в виде [-]ddd.ddd, где d-одна или несколько десятичных цифр. Число цифр после десятичной точки зависит от точности |
| g | double | Число со знаком в формате ”f” или ”e”. Используется наиболее компактная форма. Отсутствуют нули в младших разрядах. Нет десятичной точки, если за ней отсутствуют цифры |
| G | double | То же что и “g”, за исключением того, что перед экспонентой стоит “G” |
| i | int | Десятичное целое со знаком |
| n | Указатель на целое | Количество символов, записанных в поток или буфер. Адрес буфера, определяемого целочисленным аргументом |
| o | Int | Восьмеричное число без знака |
| p | Указатель на void | Печатается адрес(определяемый аргументом) |
| s | строка | printf – определяет строку однобайтных символов  wprintf - определяет строку широких символов(печать в NULL или с максимальной точностью) |
| S | строка | printf – определяет строку широких символов  wprintf - определяет строку однобайтных символов (печать в NULL или с максимальной точностью) |
| u | строка | Десятичное целое без знака |
| x | int | Шестнадцатеричное целое без знака (используются символы нижнего регистра) |
| X | int | Шестнадцатеричное целое без знака (используются символы верхнего регистра) |

Таблица 11.2 Символы преобразования для функций printf() и fprintf() (продолжение)

|  |  |
| --- | --- |
| **Флаг** | **Значение** |
| **#** | Перед ненулевыми значениями ставиться 0, 0x, или 0X (по умолчанию пробелы отсутствуют) |
| **.** | Для форматов “e”, “E”, или “f” символ “#” указывает на наличие в выходном значении десятичной точки во всех случаях (по умолчанию точка появляется тогда, когда за ним следуют цифры) |
| **-** | Если число введенных символов оказывается меньше указанного, результаты выравниваются по левому краю поля вывода(по умолчанию принято правостороннее выравнивание |
| **+** | При вводе знаковых чисел знак отображается всегда(по умолчанию знак устанавливается только перед отрицательными числами) |
| **0** | Если значению поля ширина предшествует символ ‘0’, выводимое число дополняется ведущими нулями до минимальной ширины поля вывода(по умолчанию в качестве заполнителей применяются пробелы); при левостороннем выравнивании игнорируется |
| **пробел** | Если выводится положительное знаковое число, перед ним ставится пробел(по умолчанию пробел в таких случаях не ставится); игнорируется при наличии флага + |

Поле спецификации является символом или числом, которое задает опцию форматирования. Простейший случай — знак процента и тип: например, %f. Поле "тип" используется для того, чтобы определить, что аргумент должен интерпретироваться как символ, строка или число. Поле "флаги" используется для управления печатью знаков, пробелов, десятичных точек, основания системы счисления выходных значений и так далее. Поле "ширина" указывает минимальное число печатаемых символов. Поле "точность" указывает максимальное число символов, печатаемых при выводе. Символы h | l | L являются необязательными префиксами, определяющими размер аргумента.

### Использование функций printf() и fprintf().

Мы уже обсуждали функцию printf() довольно основательно. Подобно puts(), она использует указатель строки в качестве аргумента. Функция printf() менее удобна, чем puts(), но более гибка. Разница заключается в том, что printf() не выводит автоматически каждую строку текста с новой строки. Вы должны указать, что хотите выводить с новых строк. Так,

printf(" %s\n" , string);

дает то же самое, что и

puts(string);

Вы можете видеть, что первый оператор требует ввода большего числа символов и большего времени при выполнении на компьютере. С другой стороны, printf () позволяет легко объединять строки для печати их в одной строке. Например:

printf(" Хорошо, %s, %s\n", name, MSG);

объединяет " Хорошо" с именем пользователя и с символьной строкой MSG в одну строку.

Функции ввода-вывода fprintf() и fscanf() работают почти как printf() и scanf(), но им нужен дополнительный аргумент для ссылки на сам файл. Он является первым в списке аргументов. Вот пример, иллюстрирующий обращение к этим функциям:

/\* образец использования fprintf() и fscanf() \*/

#include <stdio.h>

void main()

{

FILE \*fi;

int age;

fi = fopen("sam.txt" , "r"); /\* считывание \*/

fscanf( fi, " %d", &age); /\* fi указывает на sam \*/

fclose(fi);

fi = fopen("data.txt" , "a"); /\* дополнение \*/

fprintf( fi, "sam is %d.\n", age); /\* fi указывает на data \*/

fclose(fi);

}

Заметим, что можно было использовать fi для двух различных файлов, потому что мы закрыли первый файл, прежде чем открыть второй.

В отличие от getc() и putc() эти две функции получают указатель типа FILE в качестве первого аргумента. Две другие, описанные ниже, получают его в качестве последнего аргумента.

В следующем примере описываются переменные четырех типов: символьная, массив символов, целочисленная и вещественная; затем показано, как для каждой переменной использовать соответствующие средства управления форматированием. Исходный текст программы подробно прокомментирован, и для максимальной простоты использована нумерация строк вывода, связывающая выводимое значение и оператор, обеспечивающий это значение:

/\*Программа на С, демонстрирующая развитые средства преобразования

и форматирования\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming\Universal\_HederFile.h"

void StopWait(void);

main()

{

char c = 'A',

psz1[] = "In making a living today many no ",

/\* Для того, чтобы жить, сегдня многие \*/

psz2[] = "longer leave any room for life.";

/\* не ыыходят из комнат в течение всей жизни. \*/

int iln = 0,

ivalue = 1234;

double dPi = 3.14159265;

/\* преобразования \*/

printf("\n[%2d] %c",++iln,c); /\*01\*//\* печать символа с \*/

printf("\n[%2d] %d",++iln,c); /\*02\*//\* печать ASCII-кода символа с \*/

printf("\n[%2d] %c",++iln,90); /\*03\*//\* печать символа с ASCII-кодом 90 \*/

printf("\n[%2d] %o",++iln,ivalue); /\*04\*//\* печать ivalue в восьмеричном виде \*/

printf("\n[%2d] %x",++iln,ivalue); /\*05\*//\* печать шестнадцатеричного числа буквами нижнего регистра \*/

printf("\n[%2d] %X",++iln,ivalue); /\*06\*//\* печать шестнадцатеричного числа буквами верхнего регистра \*/

/\* опции преобразования и форматирования \*/

printf("\n[%2d] %c",++iln,c); /\*07\*//\* минимальная ширина 1\*/

printf("\n[%2d] %5c",++iln,c); /\*08\*//\* минимальная ширина 5, выравнивание вправо \*/

printf("\n[%2d] %-5c",++iln,c); /\*09\*//\* минимальная ширина 5, выравнивание влево \*/

printf("\n[%2d] %s",++iln,psz1); /\*10\*//\* 33 не-null символа с автоматическим определением ширины \*/

printf("\n[%d] %s",++iln,psz2); /\*11\*//\* 31 не-null символ с автоматическим определением ширины \*/

printf("\n[%d] %5s",++iln,psz1); /\*12\*//\* минимальная ширина 5 переопределена, автоматическая — 33 \*/

printf("\n[%d] %38s",++iln,psz1);/\*13\*//\* минимальная ширина 38, выравнивание вправо \*/

printf("\n[%d] %-38s",++iln,psz2);/\*14\*//\* минимальная ширина 38, выравнивание влево \*/

printf("\n[%d] %d",++iln,ivalue);/\*15\*//\* ширина ivalue по умолчанию 4 \*/

printf("\n[%d] %+d",++iln,ivalue);/\*16\*//\* печать ivalue со знаком "+" \*/

printf("\n[%d] %3d",++iln,ivalue);/\*17\*//\* минимальная ширина 3 переопределена, автоматическая — 4 \*/

printf("\n[%d] %10d",++iln,ivalue);/\*18\*//\* минимальная ширина 10, выравнивание вправо \*/

printf ("\n[%d] %-10d",++iln,ivalue); /\*19\*//\* минимальная ширина 10, выравнивание влево \*/

printf ("\n[%d] %010d", ++iln, ivalue) ; /\*20\*//\* выравнивание влево с дополнением пробелов \*/

printf("\n[%d] %f",++iln,dPi); /\*21\*//\* использование количества цифр по умолчанию \*/

printf("\n[%d] %20f",++iln,dPi); /\*22\*//\* минимальная ширина 20, выравнивание вправо \*/

printf("\n[%d] %020f",++iln,dPi); /\*23\*//\* выравнивание вправо с ведущими нулями \*/

printf("\n[%d] %-20f",++iln,dPi); /\*24\*//\* минимальная ширина 20, выравнивание влево \*/

/\* дополнительная точность при форматировании \*/

printf("\n[%d] %19.19s",++iln,psz1); /\*25\*//\* минимальная ширина 19, печать всех 19 \*/

printf("\n[%d] %.2s",++iln,psz1); /\*26\*//\* печать первых 2-х символов \*/

printf("\n[%d] %19.2s",++iln,psz1); /\*27\*//\* печать 2-х символов, выравнивание вправо \*/

printf("\n[%d] %-19.2s",++iln,psz1); /\*28\*//\* печать 2-х символов, выравнивание влево \*/

printf("\n[%d] %\*.\*s",++iln,19,6,psz1); /\*29\*//\* использование аргументов printf \*/

printf("\n[%d] %10.8f",++iln,dPi); /\*30\*//\* ширина 10; 8 справа от '.' \*/

printf("\n[%d] %20.2f",++iln,dPi); /\*31\*//\* ширина 20; 2 до правой границы выравнивания \*/

printf("\n[%d] %-20.4f",++iln,dPi); /\*32\*//\* 4 десятичных позиций, выравнивание влево \*/

printf("\n[%d] %20.4f",++iln,dPi); /\*33\*//\* 4 десятичных позиций, выравнивание вправо \*/

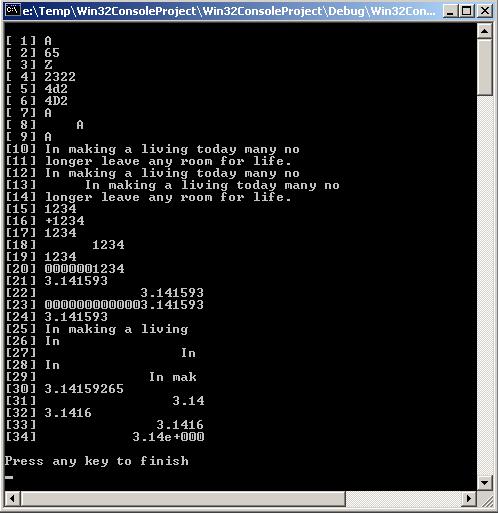
printf("\n[%d] %20.2e",++iln,dPi); /\*34\*//\* ширина 20, экспоненциальное представление \*/

StopWait(); /\* Wait a little \*/

return (0);

}

Результат работы программы выглядит следующим образом:



Изучив приведенный пример, вы сможете легко форматировать выходные данные вашей программы и выбирать те комбинации, которые соответствуют типам данных вашего приложения.

## Использование функций fread и fwrite, fseek(), ftell() и rewind().

На самом деле не все файлы выглядят как текст. В файле могут быть записаны числовые данные.

size\_t fread(void \*ptr, size\_t size, size\_t nelts, FILE \*f);

void \*ptr - указатель на ту область памяти, в которую мы читаем;

size\_t size - размер элемента, который мы читаем;

size\_t nelts - максимальное количество элементов, которые можно записать;

FILE \*f - файл, из которого читаем;

size\_t fread() - сама функция возвращает количество элементов, которые удалось прочитать.

Есть парная функция:

size\_t fwrite(const void \*ptr, size\_t size, size\_t nelts, FILE \*F);

Аналогично fread эта функция возвращает количество элементов, которые удалось записать. Тут параметр nelts просто показывает, сколько элементов надо вывести.

Когда файл открывается для записи или чтения, с ним связывается структура FILE, определенная в заголовочном файле <stdio.h>. Эта структура связывает с каждым открытым файлом счетчик положения текущей записи. Сразу после открытия файла его значение равно 0. Каждая операция вызывает приращение значения этого счетчика на число записанных или прочитанных из файла байтов. Функции позиционирования — fseek(), ftell() и rewind() позволяют изменять или получать значение счетчика, связанного с файлом.

**Функция ftell()** возвращает текущее значение счетчика связанного с файлом. Она имеет следующий прототип:

long int ftell(FILE \*stream);

В случае ошибки она возвращает -1L.

**Функция fseek()** имеет следующий прототип:

int fseek(FILE \*stream, long offset,

int from);

Эта функция изменяет позиционирование файлового потока stream (изменяя значение указанного счетчика) на offset относительно позиции, определяемой параметром from. Для потоков в текстовом режиме параметр offset должен быть равен 0 или значению, возвращаемому функцией ftell(). Параметр from может принимать следующие значения:

SEEK\_SET (=0) начало файла;

SEEK\_CUR (=1) текущая позиция в файле;

SEEK\_END (=2) конец файла.

Функция возвращает 0, если указатель текущей позиции в файле успешно изменен, и отличное от нуля значение в противном случае.

**Функция rewind()** имеет следующий прототип:

void rewind(FILE \*stream);

Она устанавливает файловый указатель позиции в начало потока.

Рассмотрим пример, демонстрирующий работу этих функций:

#include <string.h>

#include <stdio.h>

using namespace std;

struct Client{

int Num;

char SurName[27];

char Name[21];

char SecName[21];

};

int main(void)

{

setlocale(LC\_ALL, "Rus");

FILE \*stream;

Client AClient, RClient;

long int pos;

//Открываем файл для вывода

if ((stream = fopen("SAMPLE.DAT", "wb")) == NULL){

printf("Cannot open output file.\n");

getchar(); getchar();

return 1;

}

AClient.Num = 1;

strcpy(AClient.SurName,"Petrov");

strcpy(AClient.Name, "Petr");

strcpy(AClient.SecName, "Petrovich");

//Запись структуры в файл

fwrite (&AClient, sizeof (AClient), 1, stream);

pos = ftell(stream);

//Выводим позицию файла и длину структуры

printf("The file pos = %d "

"structure length = %d\n", pos, sizeof(AClient));

//Репозиционируем файл

rewind(stream);

//Открываем файл для чтения

if ((stream = fopen("SAMPLE.DAT", "rb"))== NULL){

printf("Cannot open input file.\n");

getchar(); getchar();

return 2;

}

fread (&RClient, sizeof (RClient), 1, stream);

//Закрываем файл

fclose(stream);

printf("The structure contains:\n");

printf("Num = %d SurName = %s Name = %s"

"SecName = %s",

RClient.Num, RClient.SurName,

RClient.Name, RClient.SecName);

getchar(); getchar();

return 0;

}



Этот пример представляет собой модификацию предыдущего примера. Здесь файл открывается лишь однажды, и после записи в него структуры на экран выводится значение указателя позиции и длины структуры. При выполнении программа выводит на экран:

The file pos = 76 structure length = 76

The structure contains:

Num = 1 SurName = Petrov Name = Petr

SecName = Petrovich

Функция fseek() позволяет нам обрабатывать файл подобно массиву и непосредственно достигать любого определенного байта в файле, открытом функцией fорen().

**fseek и ftell**

Для файлов, которые открыты на чтение есть полезные функции. Одна из них это:

int fseek(FILE \*f, long offset, int flag);

* FILE \*f - файл, в котором передвигаемся;
* long offset - количество байтов для отступа, отступ производится в соответствии с 3-м параметром;
* int flag - позиция, от которой будет совершен отступ; в стандартной библиотеке C для этого параметра определены 3 константы:

SEEK\_SET -- начало файла;

SEEK\_CUR -- текущас позиция;

SEEK\_END -- конец файла;

int fseek() -- сама функция возвращает ноль, если операция прошло успешно, иначе возвращается ненулевое значение.

Еще одна полезная функция может определить текущее положение в файле (который открыт для чтения):

long int ftell(FILE \*f);

Вот простой пример, показывающий, как она работает. Как и в наших предыдущих примеpax, функция использует аргумент командной строки для получения имени файла, с которым она работает. Заметим, что fseek() имеет три аргумента и возвращает значение типа int.

/\* использование fseek() для печати содержимого файла \*/

#include <stdio.h>

void main(int number,char \*names []) /\* не следует использовать argc и argv \*/

{

FILE \*fp;

long offset = 0L; /\* обратите внимание, что это тип long \*/

if ( number < 2)

puts ("I need a name of file as an argument");

else

{

if ((fp = fopen(names[1], "r")) == 0)

printf(" I can't open file %s.\n" , names[1]);

else

{

while( fseek(fp, offset++ , 0) == 0)

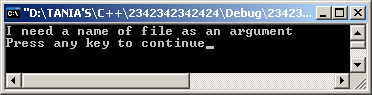
putchar(getc(fp) );

fclose(fp);

}

}

}



Первый из трех аргументов функции fseek() является указателем типа FILE на файл, в котором ведется поиск. Файл следует открыть, используя функцию fopen().

Второй аргумент назван "offset" (вот почему мы выбрали данное имя для переменной). Этот аргумент сообщает, как далеко следует передвинуться от начальной точки (см. ниже); он должен иметь\* значение типа long, которое может быть положительным (движение вперед) или отрицательным (движение назад).

Третий аргумент является кодом, определяющим начальную точку:

**Код Положение в файле**

0 начало файла

1 текущая позиция

2 конец файла

Функция fseek() возвращает 0, если все хорошо, и -1, если есть ошибка, например попытка перемещаться за границы файла.

Теперь мы можем разъяснить наш маленький цикл:

while ( fseek(fp, offset ++ , 0) == 0)

putchar(getc(fp) );

Поскольку переменная offset инициализирована нулем, при первом прохождении через цикл мы имеем выражение

fseek(fp, 0L, 0)

означающее, что мы идем в файл, на который ссылается указатель fp, и находим байт, отстоящий на 0 байт от начала, т. е. первый байт. Затем функция putchar() печатает содержимое этого байта. При следующем прохождении через цикл переменная offset увеличивается до 1L, печатается следующий байт. По существу, переменная offset действует подобно индексу для элементов файла. Процесс продолжается до тех пор, пока offset не попытается попасть в fseek() после конца файла. В этом случае возвращает значение — 1 и цикл прекращается.

Этот последний пример чисто учебный. Нам не нужно использовать fseek(), потому что getc() так или иначе проходит через файл байт за байтом; fseek() приказала getc() «посмотреть» туда, куда она сама уже собиралась посмотреть.

Вот пример, в котором выполняется что-то несколько более необычное (Мы благодарим Вильяма Шекспира за этот пример в пьесе «Двенадцатая ночь»).

/\* чередование печати в прямом и обратном направлениях \*/

#include <stdio.h>

void main(int number,char \*names[] ) /\* вам не нужно применять argc и argv \*/

{

FILE \*fp;

long offset = 0L;

if ( number < 2)

puts(" Мне нужно имя файла в качестве аргумента.");

else

{

if ( (fp = fopen(names[1], "r")) ==0)

printf(" Я не могу открыть %s.\n", names[1]);

else

{

while ( fseek(fp, offset ++ , 0) == 0)

{

putchar(getc(fp));

if ( fseek(fp,-(offset + 3), 2) == 0)

putchar(getc(fp));

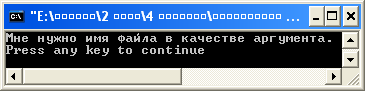
}

fclose(fp);

}

}

}



Применение этой программы к файлу, содержащему имя «Мальволио», дает такой приятный результат:

МоаилльоввоьллиаоМ

Наша программа печатает первый символ файла, затем последний, затем второй, затем предшествующий последнему и т. д. Мы только добавили вот эти строки в последнюю программу:

if ( fseek(fp,-(offset + 3), 2) == 0)

putchar(getc(fp));

Код 2 в операторе предполагает, что мы будем считать позиции от конца файла. Знак минус означает счет в обратном направлении. + 3 стоит здесь потому, что мы начинаем с последнего регулярного символа файла и пропускаем несколько символов «новая строка» и EOF в самом конце файла. (Точное значение этой корректировки зависит от типа системы. Наши файлы имеют в конце по два символа новой строки, за которыми следуют два EOF, поэтому мы как раз их и обходим.)

Таким образом, эта часть программы чередует печать в обратном направлении и печать в прямом направлении. Следует заметить, что в некоторых системах может не предусматриваться код 2 для fseek().

Функции fseek(), ftell() и rewind() можно использовать для определения или изменения местоположения маркера файла. Функция fseek() смещает маркер файла, на который указывает pf, на число ibytes байтов либо относительно начала файла (ifrom=()), либо относительно текущего положения маркера (ifrom=1), либо относительно конца файла (ifrom=2). В С имеется три предопределенных константы, которые можно использовать вместо переменной ifrom: SEEK\_SET (смещение относительно начала файла), SEEK\_CUR (текущее положение маркера) и SEEK\_END (смещение относительно конца файла). Функция fseek() возвращает ноль, если операция была успешной, и EOF — в противном случае. В общем виде синтаксис функции fseek() следующий: fseek(pf, ibytes, ifrom) ;

Функция ftell() возвращает текущее положение маркера файла, на который указывает pf. Это положение представляет собой смещение в байтах от на-чала файла. Синтаксис функции ftell() следующий:

long\_variable=ftell(pf);

Значение, возвращаемое ftell(), можно использовать при последующем вызове функции fseek().

Функция rewind() просто устанавливает маркер файла, на который ссылается pf, в начало этого файла. Синтаксис функции rewind() выглядит следующим образом:

rewind(pf);

В следующей программе на С иллюстрируется использование функций fseek(), ftell() и rewind():

/\*Программа на С, иллюстрирующая использование функций fseek,

ftell и rewind\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming\TestProg\Universal\_HederFile.h"

void StopWait(void);

main()

{

FILE \*pf;

char c;

long llocation;

pf=fopen("E:\\LECTURE\\AlgorithmProgramming\\TestProg\\test.dat","r+");

c=fgetc(pf);

putchar(c);

c=fgetc(pf);

putchar(c);

llocation=ftell(pf);

c=fgetc(pf);

putchar(c);

fseek(pf,llocation,0);

c=fgetc(pf);

putchar(c);

fseek(pf,llocation,0);

fputc('E',pf);

fseek(pf,llocation,0);

c=fgetc(pf);

putchar(c);

rewind(pf);

c=fgetc(pf);

putchar(c);

StopWait(); /\* Wait a little \*/

return (0);

}

По определению переменная llocation имеет тип long. Это вызвано тем, что С поддерживает файлы длиннее, чем 64К. Входной файл TEST.DAT содержит строку "ABCD". После того как программа открывает этот файл, при первом обращении к функции fgetc() считывается символ "А", который затем печатается на дисплее. Следующая пара операторов считывает и печатает символ "В".

После вызова функции ftell() переменная llocation становится равной текущему положению маркера файла. Оно измеряется как смещение в байтах от начала файла. Поскольку символ "В" уже обработан, llocation содержит 2. Это означает, что маркер файла указывает на третий символ, который на 2 байта отстоит от первого символа — "А".

После этого другая пара операторов ввода/вывода считывает и печатает на дисплее символ "С". После выполнения этих операторов маркер файла смещен на 3 байта от начала файла и указывает на четвертый символ — "D".

В этой точке программы вызывается функция fseek(). Она должна сместить маркер на хранящееся в llocation число байт (то есть на 2 байта) относительно начала файла (поскольку третий параметр функции fseek() — ноль; это объяснялось выше). В результате этой операции маркер файла устанавливается на третий символ в файле. Переменной с опять присваивается значение символа "С", которое печатается во второй раз.

При повторном вызове функции fseek() используются те же параметры, что и при первом. Функция fseek() устанавливает маркер на третий символ —

"С" (смещение 2 байта от начала). Однако следующий оператор не читает символ "С" в третий раз, а записывает вместо него новый символ — "Е". Поскольку после этого маркер смещается за это новое значение, функция fseek() вызывается еще раз для того, чтобы проверить запись этого символа в файл.

Следующая пара операторов считывает новое значение, "Е", и печатает его на дисплее. После этого программа вызывает функцию rewind(), которая снова перемещает указатель pf к началу файла. Затем, когда вызывается функция fgetc(), считывается символ "А" и печатается на экране. Результат работы программы выглядит так:

АВССЕА

Эти же принципы, проиллюстрированные на простом примере, можно использовать для создания файла записей со свободным доступом. Пред положим, что для записи в файл персонала имеется следующая информация: номер социальной страховки, имя и адрес. Допустим, что для номера страховки выделено 11 символов в формате ddd-dd-dddd, а имя и адрес занимают дополнительно 60 символов (или байт). Тогда каждая запись будет иметь длину 11 + 60 = 71 байт.

Заполненными могут быть не все возможные последовательные записи в дисковом файле со свободным доступом; нужно, чтобы запись имела некий флаг, указывающий на использование данной дисковой записи. Для этого к записи нужно добавить еще один байт и 2 дополнительных байта для номера записи; длина записи для одного человека в итоге составит 74 байта. Отдельная запись может выглядеть следующим образом:

1 U111-22-3333Linda Lossannie, 521 Alan Street, Anywhere, USA

Запись номер 1 в файле будет занимать байты с нулевого по 73-й; запись 2 занимает байты с 74 по 147; запись 3 занимает байты с 148 по 221; и так далее. Если использовать номер записи в сочетании с функцией fseek(), то на диске можно найти любую запись. Например, для того чтобы найти начало записи 2, можно использовать следующие операторы:

loffset=(iwhich\_record - 1) \* sizeof(stA\_PERSON); fseek(pfi,loffset,0);

После того как маркер перемещен к началу выбранной записи, информация из этой записи может считываться или обновляться при помощи раз-личных функций ввода/вывода, подобных fread() и fwrite().

За исключением символов комментария /\* и \*/ и заголовочного файла stdio.h описанная программа может работать и в C++. Достаточно заменить символом // оба разделителя /\* и \*/ , а файл stdio.h заменить на iostream.h.

В заключение заметим, что библиотека стандартных функций ввода-вывода содержит много других, не рассмотренных здесь функций.

### Использование встроенного отладчика.

Попробуйте ввести следующую программу и после запроса на поиск 25-й записи распечатать значение переменной stcurrent\_person.irecordnum:

/\*Программа на С, обрабатывающая файл с произвольным доступом при

помощи функций fseek, fread и fwrite\*/

#include "stdafx.h"

#include "E:\LECTURE\AlgorithmProgramming\TestProg\Universal\_HederFile.h"

void StopWait(void);

#define iFIRST 1

#define iLAST 50

#define iSS\_SIZE 11

#define iDATA\_SIZE 60

#define cVACANT 'V'

#define cUSED 'U'

typedef struct strecord

{

int irecordnum;

char cavailable; /\* V - свободна, U - используется \*/

char csoc\_sec\_num[iSS\_SIZE];

char cdata[iDATA\_SIZE ];

} stA\_PERSON;

main()

{

FILE \*pfi;

stA\_PERSON stcurrent\_person;

int i,iwhich\_record;

long int loffset;

pfi=fopen("E:\\LECTURE\\AlgorithmProgramming\\TestProg\\sample.fil","r+");

for(i = iFIRST; i <= iLAST; i++)

{

stcurrent\_person.cavailable=cVACANT;

stcurrent\_person.irecordnum=i;

fwrite(&stcurrent\_person,sizeof(stA\_PERSON),1,pfi);

}

/\* Пожалуйста, введите номер записи для поиска \*/

printf("Please enter the record you would like to find.");

/\* "Возможны значения от 1 до 50:" \*/

printf("\nYour response must be between 1 and 50: ");

scanf("%d",&iwhich\_record);

loffset=(iwhich\_record - 1) \* sizeof(stA\_PERSON);

fseek(pfi,loffset,0);

fread(&stcurrent\_person,sizeof(stA\_PERSON),1,pfi);

fclose (pfi) ;

StopWait(); /\* Wait a little \*/

return (0);

}

Ключевое слово typedef определяет stA\_PERSON как структуру, имеющую следующие поля: 2-байтный номер записи irecordnum, однобайтный символьный флаг доступности записи cavailable, 11-байтный массив символов для хранения номера страховки csoc\_sec\_num и 60-байтное поле данных cdata. В результате общий размер структуры равен 2 + 1 + 11 + 60 = 74 байтам.

После того как программа открыла текстовый файл в режиме "считывание/запись", она создает и запоминает 50 записей, каждая из которых имеет уникальный номер irecordnum; все записи помечены как "свободные" — cVACANT. Для выполнения оператора fwrite() нужно знать адрес записываемой структуры, размер в байтах записываемой информации, количество выводимых блоков и указатель на файл вывода. После всего этого программа запрашивает у пользователя номер записи, которую он хочет найти.

Поиск записи осуществляется в два этапа. Во-первых, нужно вычислить смещение относительно начала файла. Например, первая запись запоминается в байтах с нулевого по 73-й, вторая — с 74 по 148 и так далее. После вычитания единицы из номера записи, введенного пользователем, программа умножает это значение на число байтов, занимаемых каждой структурой и определяет значение смещения loffset. Например, для нахождения второй записи выполняются следующие вычисления: (2-1) х 74. В результате получается, что вторая запись начинается со смещением 74. После вычисления этого значения выполняется вызов функции fseek(), которая перемещает маркер файла на loffset байт.

Пока выполняется трассировка программы при поиске записей от 1 до 10, все идет нормально. Однако, что происходит, когда вы просите найти одиннадцатую запись? Получается "мусор". Причина этого кроется в том, что программа открыла файл в текстовом режиме. Записи с 1 по 9 точно равны 74 байтам, однако записи с 10-й и далее имеют длину 75 байт. Следовательно, десятая запись имеет соответственно вычисленное смещение loffset, однако в файле занимает на один байт больше. Поэтому одиннадцатая запись начинается с адреса, полученного в результате следующего мо-дифицированного выражения:

loffset=((iwhich\_record - 1) \* sizeof(stA\_PERSON)) + 1;

Однако эти вычисления не будут работать с первыми девятью записями. Решение проблемы — открыть файл в двоичном режиме:

pfi=fopen("А:\\sample.fil","r+b");

В текстовом режиме любое число из двух цифр трактуется как два отдельных символа, в результате чего записи с номерами из двух цифр удлиняются на 1. В двоичном режиме целое значение номера записи интерпретируется правильно. Нужно обращать внимание на режим открываемого для ввода/вывода файла.

## Форматированный ввод.

В программах на С форматированный ввод обеспечивают весьма гибкие функции scanf() и fscanf(). Главное отличие между ними заключается в том, что для последней функции необходимо явно указывать входной файл, из которого считываются данные. В табл. 11.3 перечислены все возможные управляющие символы, которые можно использовать с функция-ми scanf(), fscanf() и sscanf().

Таблица 11.3. Управляющие символы для функций scanf(), fscanf() и sscanf()

|  |  |  |  |
| --- | --- | --- | --- |
| **Символ** | **Представление** | **Пример ввода** | **Тип принимающего адресного параметра** |
| c | Символ | W | char |
| s | Строка | William | char |
| d | int | 23 | int |
| hd | short | -99 | short |
| id | long | 123456 | long |
| o | octat | 1727 | int |
| ho | short восьмеричное | 1727 | short |
| io | long восьмеричное | 1727 | long |
| x | шестнадцатеричное | 265 | int |
| hx | short шестнадцатеричное | 265 | short |
| tx | long шестнадцатеричное | 265 | long |
| e | float как float | 3.14159e+03 | float |
| f | То же, что “e” |  | double |
| ie | float как double | 3.14159e+03 | double |
| if | То же, что “ie” |  |  |
| [A-Za-z] | Строки из одних символов | Test string | char |
| [0-9] | Строки из одних цифр | 098231345 | char |

## Использование функций scanf(), fscanf() и sscanf().

Все три функции, scanf(), fscanf() и sscanf(), можно использовать для ввода чрезвычайно сложных данных. Взгляните, например, на следующий оператор:

scanf("%2d%5s%4f",&ivalue,psz,&fvalue);

Этот оператор вводит целое число из двух цифр, строку из пяти символов и вещественное число, занимающее максимально четыре позиции (2.97, 12.5 и так далее). Сможете ли вы определить, что выполняет следующий оператор:

scanf ("%\*[ \t\n] \"%[^A-Za-z] %[^\"] \"",ps1,ps2);

Вначале оператор считывает, но не запоминает любой пустой символ (пробел). Это выполняется при помощи следующей спецификации форма-та: "%\*[ \t\n]". Символ (\*) указывает функции, что она должна получить указанные данные, но не запоминать их ни в одной переменной. Если во входной строке содержатся только символы пробела, табуляции или перевода строки, то функция scanf() продолжает считывание до тех пор, пока не встретит двойную кавычку ("). Для этого используется спецификатор формата \", который указывает на то, что введенное значение должно совпадать с заданным символом. Символ двойной кавычки, однако, не вводится. огда функция scanf() обнаружила двойную кавычку, она получает указание вводить все символы, являющиеся цифрами, в переменную ps1. Для этого в спецификации формата %[^A-Za-z] используется модификатор — знак вставки (^), который указывает на то, что можно вводить любые символы, за исключением заглавных букв от "А" до "Z" и строчных букв от "а" до "z". Если бы знака вставки не было, то строка должна была бы содержать только буквенные символы. Знак тире между двумя символами "А" и "Z" и "а" и "z" говорит о том, что нужно рассматривать весь диапазон значений.

Мы уже использовали ранее функцию scanf() и формат %s для считывания строки. Основное различие между scanf() и gets() заключается в том, как они определяют, что достигли конца строки; scanf () предназначена скорее для получения слова, а не строки. Функция gets (), как мы уже видели, принимает все символы до тех пор, пока не встретит первый символ «новая строка». Функция scanf () имеет два варианта. Для любого из них строка начинается с первого встретившегося непустого символа. Если вы используете формат %s, строка продолжается до (но не включая) следующего пустого символа (пробел, табуляция или новая строка). Если вы определяете размер поля как %10s, то функция scanf() считает не более 10 символов или же считает до любого пришедшего первым пустого символа.

Функция scanf () возвращает целое значение, равное числу считанных символов, если ввод прошел успешно, или символ EOF, если он встретился.

/\* scanf () и подсчет количества \*/

#include <stdio.h>

void main ()

{

static char name1[40], name2[11];

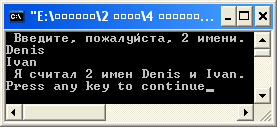
int count;

printf(" Введите, пожалуйста, 2 имени.\n");

count = scanf(" %s %10s", name1, name2);

printf(" Я считал %d имен %s и %s.\n", count, name1, name2);

}



Вот два примера работы программы:

Введите, пожалуйста, два имени.

Джессика Джукс

Я считал 2 имени Джессика и Джукс.

Введите, пожалуйста, 2 имени.

Лиза Апплеботтхэм

Я считал 2 имени Лиза и Апплеботтх.

Во втором примере были считаны только первые 10 символов от Апплеботтхэм, так как мы использовали формат %10s.

Если вы получаете только текст с клавиатуры, лучше применять функцию gets(). Она проще в использовании, быстрее и более компактна. Функция scanf () предназначена в основном для ввода смеси типов данных в некоторой стандартной форме. Например, если каждая вводимая строка содержит наименование инструмента, количество его на складе и стоимость каждого инструмента, вы можете использовать функцию scanf () или можете создать свою собственную функцию, которая выполняет проверку некоторых ошибок при вводе.

Следующая спецификация формата, %[^\"], сообщает функции ввода, что необходимо читать все оставшиеся символы, исключая двойные кавычки, в переменную ps2. Последняя спецификация, \", указывает на то, что строка должна закончиться двойной кавычкой. Одни и те же символы форматирования ввода можно использовать с функциями fscanf() и sscanf(). Различие между функциями scanf() и fscanf() заключается лишь в том, что для последней необходимо указывать файл ввода. Функция sscanf() аналогична scanf(), за исключением того, что она читает данные из массива, а не из файла.

Следующий пример показывает, как можно использовать функцию sscanf() для преобразования строки (состоящей из цифр) в целое число. Если ivalue имеет тип int, a psz является массивом типа char, содержащим строку цифр, то при помощи следующего оператора можно преобразовать строку psz в тип int и запомнить его в переменной ivalue:

sscanf(psz,"%d",&ivalue);

Очень часто функции gets() и sscanf() используются совместно, так как функция gets() читает целую входную строку, а функция sscanf(0 просматривает строку и преобразует ее согласно спецификациям формата.

Часто при работе с функцией scanf(), если программисты пытаются использовать ее в сочетании с другими функциями символьного ввода, такими как getc(), getch(), getchar(), gets() и так далее, возникает одна проблема. Типичная последовательность действий выглядит так: функция scanf() используется для ввода символов и преобразования их в данные других типов. Затем используется какая-нибудь функция символьного ввода, например, getch(), и оказывается, что она работает не так как ожидалось. Это происходит потому, что иногда функция scanf() считывает не все предназначенные для ввода данные, а оставшиеся (не введенные) данные ошибочно считываются другими функциями ввода (в том числе и следующей функцией scanf()). Если вы используете функцию scanf() в программе, то в целях безопасности не используйте в той же программе другие функции ввода.

## Распределение памяти: malloc() и calloc().

Ваша программа должна предоставить достаточный объем памяти для запоминания используемых данных. Некоторые из этих ячеек памяти распределяются автоматически. Например, мы можем объявить

char place [] = " Залив Свиной печенки" ;

и будет выделена память, достаточная для запоминания этой строки.

Или мы можем быть более конкретны и запросить определенный объем памяти:

int plates[100];

Это описание выделяет 100 ячеек памяти, каждая из которых предназначена для запоминания целого значения.

Язык Си не останавливается на этом. Он позволяет вам распределять дополнительную память во время работы программы. Предположим, например, вы пишете диалоговую программу и не знаете заранее, сколько данных вам придется вводить. Можно выделить нужный вам (как вы считаете) объем памяти, а затем, если понадобится, потребовать еще. Ниже дан пример, в котором используется функция malloc(), чтобы сделать именно это. Кроме того, обратите внимание на то, как такая программа применяет указатели.

/\* добавляет память, если необходимо \*/

#include <stdio.h>

#define STOP /\* сигнал прекращения ввода \*/

#define BLOCK 100 /\* байты памяти \*/

#define LIM 40 /\* предельная длина вводимой строки \*/

#define MAX 50 /\* максимальное число вводимых строк \*/

#define DRAMA 20000 /\* большая задержка времени \*/

main()

{

char store[BLOCK]; /\* исходный блок памяти \*/

char symph[LIM]; /\* приемник вводимых строк \*/

char \*end; /\* указывает на конец памяти \*/

char \*starts[MAX]; /\* указывает на начала строк \*/

int index = 0; /\* количество вводимых строк \*/

int count; /\* счетчик \*/

char \*malloc(); /\* распределитель памяти \*/

starts [0] = store;

end = starts [0] + BLOCK - 1;

puts(" Назовите несколько симфонических оркестров." );

puts(" Вводите по одному: нажмите клавишу [ввод] в начале");

puts(" строки для завершения вашего списка. Хорошо, я готова.");

while( strcmp(fgets(symph,LIM, stdin), STOP) != 0 && index < MAX)

{

if ( strlen(symph) > end - starts [index])

{ /\* действия при недостатке памяти для запоминания вводимых данных\*/

puts(" Подождите секунду. Я попробую найти дополнительную память.");

starts [index] = malloc(BLOCK);

end = starts [index] + BLOCK - 1;

for( count = 0; count < DRAMA; count++ );

puts(" Нашла немного!" );

}

strcpy(starts[index], symph);

starts [index + 1] = starts [index] + strlen(symph) + 1;

if (++index < MAX)

printf("Это %d. Продолжайте, если хотите\n", index);

}

puts(" Хорошо, вот что я получила:");

for( count = 0; count < index; count++)

puts(starts [count]);

}

Вот образец работы программы:

Назовите несколько симфонических оркестров.

Вводите их по одному; нажмите клавишу [ввод] в начале

строки для завершения вашего списка. Хорошо, я готова.

**Сан-францисский симфонический**

Это 1. Продолжайте, если хотите.

**Чикагский симфонический**

Это 2. Продолжайте, если хотите.

**Берлинский филармонический**

Это 3. Продолжайте, если хотите.

**Московский камерный**

Это 4. Продолжайте, если хотите.

**Лондонский симфонический**

Это 5. Продолжайте, если хотите.

**Венский филармонический**

Подождите секунду. Я попробую найти дополнительную память

Нашла немного!

Это 6. Продолжайте, если хотите.

**Питтсбургский симфонический**

Это 7. Продолжайте, если хотите.

Хорошо, вот что я получила:

Сан-францисский симфонический

Чикагский симфонический

Берлинский филармонический

Московский камерный

Лондонский симфонический

Венский филармонический

Питтсбургский симфонический

Сначала давайте посмотрим, что делает функция malloc(). Она берет аргумент в виде целого без знака, которое представляет количество требуемых байтов памяти. Так, malloc(BLOCK) требует 100 байт. Функция возвращает указатель на тип char в начало нового блока памяти. Мы использовали описание

char \*malloc();

чтобы предупредить компилятор, что malloc() возвращает указатель на тип char. Поэтому мы присвоили значение этого указателя элементу массива starts [index] при помощи оператора

starts[index] = malloc(BLOCK);

Хорошо, давайте теперь рассмотрим проект программы, заключающийся в том, чтобы запомнить все исходные строки подряд в большом массиве store. Мы хотим использовать starts[0] для ссылки на начало первой строки, starts[1] — второй строки и т. д. На промежуточном этапе программа вводит строку в массив symph. Мы использовали fgets() вместо gets(), чтобы ограничить входную строку длиной массива symph.

16_01

Прежде чем копировать symph в store, мы должны проверить, достаточно ли для нее оставшегося места. Указатель end ссылается на конец памяти, а текущее значение starts[index] ссылается на начало неиспользованной памяти. Таким образом, мы можем сравнить разницу между этими двумя указателями с длиной symph и определить, достаточно ли осталось памяти.

Если места недостаточно, вызываем malloc(), чтобы подготовить дополнительную память. Мы устанавливаем starts[index] на начало нового блока памяти, a end — на конец нового блока. Заметим, что у нас нет имени этой новой памяти. Она не является, например, расширением store. У нас есть только обозначения указателей, ссылающихся на новую область памяти.

Когда программа работает, на каждую новую строку ссылается элемент массива указателей starts. Некоторые строки находятся в Store, другие — в одной или нескольких новых областях памяти.

Но пока у нас есть указатели, мы можем работать со строками, как показывает нам часть программы, выполняющая вывод на печать.

Таким образом, используется malloc(). Но предположим, что вы хотите работать с памятью типа int, а не char. Можете и здесь использовать malloc(). Вот как это делается:

char \*malloc(); /\* по-прежнему описываем как указатель на char \*/

int \*newmem;

newmem = (int \*) malloc (100); /\* используем операцию приведения типа \*/

Снова требуется 100 байт. Операция приведения типа преобразует значение, возвращенное указателем на тип char, в указатель на тип int. Если, как в нашей системе, int занимает два байта памяти, это значит, что newmem + 1 будет увеличивать указатель на два байта, т. е. передвигать его к следующему целому. Это также означает, что 100 байт можно использовать для запоминания 50 целых чисел.

Другую возможность распределения памяти дает нам применение функции calloc():

char \*calloc();

long \* newmem;

newmem = (long \*) calloc(100, sizeof(long));

Подобно malloc() функция calloc() возвращает указатель на char. Нужно использовать оператор приведения типа, если вы хотите запомнить другой тип. Эта новая функция имеет два аргумента, и оба они должны быть целыми без знака. Первый аргумент содержит количество требуемых ячеек памяти. Второй аргумент — размер каждой ячейки в байтах. В нашем случае long использует четыре байта, поэтому оператор выделит 100 четырехбайтных элементов, используя в целом 400 байтов памяти.

Применяя sizeof (long) вместо 4, мы сделали эту программу более мобильной. Она будет работать на системах, где long имеет размер, отличный от четырех.

Функция calloc() имеет еще одну особенность; она обнуляет содержимое всего блока.

Ваша библиотека языка Си, вероятно, предоставляет несколько других функций управления памятью, и вы можете захотеть проверить их.

## Другие библиотечные функции.

Большинство библиотек будут выполнять и ряд дополнительных функций в тех случаях, которые мы рассмотрели. Кроме функций, распределяющих память, есть функции, освобождающие память после работы с нею. Могут быть другие функции, работающие со строками, например такие, которые ищут в строке определенный символ или сочетание символов.

Некоторые функции, работающие с файлами, включают ореn(), close(), create(), fseek(), read() и write(). Они выполняют почти те же самые задачи, что и функции, которые мы обсудили, но на более фундаментальном уровне. Действительно, функции, подобные fopen(), обычно пишутся с применением этих более общих функций. Они немного более трудны в использовании, но могут работать с двоичными файлами так же, как и с текстовыми.

Ваша система может иметь библиотеку математических функций. Обычно такая библиотека будет содержать функции квадратного корня, степенные, экспоненциальные, различные тригонометрические функции и функцию получения случайных чисел.

Вам нужно время, чтобы освоить то, что предлагает ваша система. Если у нее нет того, что вам нужно, создайте свои собственные функции. Это часть языка Си. Если вы полагаете, что можете улучшить работу, скажем, функции ввода, сделайте это! А когда вы усовершенствуете и отшлифуете свои методы программирования, вы перейдете от обычного языка Си к блестящему языку Си.

**Что вы должны были узнать.**

* Что такое библиотека языка Си и как ее использовать.
* Как открывать и закрывать текстовые файлы: fopen() и fclose()
* Что такое тип FILE
* Как читать из файла и записывать в файл: getc(), putc(),
* fgets(), fputs(), fscanf(), fprintf()
* Как проверять классы символов: isdigit(), isalpha() и т. д.
* Как превращать строки в числа: atoi() и atof()
* Как осуществлять быстрый выход: ехit()
* Как распределять память: malloc(), calloc()

**Вопросы и ответы.**

**Вопросы.**

1. Что неправильно в этой программе?

main()

{

int \*fp;

int k;

fp = fopen(" желе" );

for (k = 0; k < 30; k++ )

fputs(fp, " Нанетта ест желе.");

fclose(" желе");

}

1. Что будет делать следующая программа?

#include <stdio.h>

#include <ctype.h>

main(argc, argv)

int argc;

char \*argv[];

{

int ch;

FILE \*fp;

if ( (fp = fopen(rag[1], " r")) == NULL)

exit(1);

while ( (ch = getc(fp)) != EOF)

if( isdigit(ch) )

putchar(ch);

fclose (fp);

}

1. Все ли правильно в выражении isalpha(c[i]), где с является массивом типа char? Что можно сказать о isalpha(c[i++])?
2. Используйте функции классификации символов для подготовки выполнения atoi().
3. Как вы могли бы распределить память для размещения массива структур?

**Ответы.**

1. Должна быть директива #include <stdio.h> для определения ее файлов. Следует описать указатель fp файла: FILE \*fp; функция fopen() должна иметь вид: fopen ("желе" , "w"), или, может быть, включать "а" . Порядок аргументов в fputs() должен быть обратным. Функция fclose() требует указателя файла, а не имени файла: fclose(fp).
2. Она будет открывать файл, заданный как аргумент командной строки, и выводить на печать все цифры в файле. Программа должна проверять (но не делает этого), не аргумент ли это командной строки.
3. Первое выражение правильно, так как c[i] имеет значение типа char. Второе выражение не выводит компьютер из строя, но может давать непредсказуемый результат. Причина в том, что isalpha() является макроопределением, у которого, по всей вероятности, аргумент появляется дважды в определяющем выражении (проверка на принадлежность к регистру строчных букв, а затем — прописных букв) и это дает в результате два увеличения i. Лучше всего избегать использования оператора увеличения в аргументе макрофункции.

#include < stdio.h>

#include < ctype.h>

#define issign(c) ( ((c) == '-' || (c) == '+') ? (1) : (0) )

atoi(s);

char \*s;

{

int i = 0;

int n, sign;

while ( isspace(s[i] ) )

i++ ; /\* пропуск пустого символа \*/

sign = 1;

if (issign(s[i]) ) /\* установка необязательного знака \*/

sign = (s[i++ ] == '+') ? 1 : -1;

for (n = 0; isdigit(s[i] ); i++ )

n = 10\*n + s[i] - '0';

return( sign \* n);

}

1. Предположим, что wine является именем структуры. Эти операторы, надлежащим образом расположенные в программе, будут выполнять данную работу.

struct wine \*ptrwine;

char \*calloc();

ptrwine = (struct wine \*) calloc (100, sizeof (struct wine));

**Упражнения.**

1. Напишите программу копирования файла, которая использует имена исходного файла и копируемого файла как аргументы командной строки.
2. Напишите программу, которая будет принимать все файлы, заданные рядом аргументов командной строки, и печатать их один за другим. Используйте argc для создания цикла.
3. Модифицируйте вашу программу инвентаризации книг так, чтобы информация, которую вы вводите, добавлялась в файл, названный my books.
4. Используйте gets( ) и atoi( ) для создания функции, эквивалентной нашей getint().
5. Перепишите нашу программу, считающую слова, используя макроопределения ctype.h и аргумент командной строки для обработки файла.

**Что вы должны были узнать.**

* Как объявить строку символов: static char fun[] и т. д.
* Как инициализировать строку символов: static char \*p0 = "0!"
* Как использовать gets () и puts ()
* Как использовать strlen(), strcmp (), strcpy() и strcat ()
* Как использовать аргументы командной строки.
* В чем сходство и различие описателей char \*bliss и char bliss []
* Как создать строковую константу: «используя кавычки».

**Вопросы и ответы.**

**Вопросы.**

1. Что неправильно в этой попытке описания символьной строки?

main ()

{

char name[] = {'F', 'e', 's', 's'};

1. Что напечатает эта программа?

#include <stdio.h>

main ()

{

static char note[] = " До встречи в буфете.”

char \*ptr,

ptr = note;

puts(ptr);

puts(++ptr);

note[7] = '\0';

puts(note);

puts(++ptr);

}

1. Что напечатает эта программа?

main ()

{

static char food[] = "Йумми";

char \*ptr;

ptr = food + strlen(food);

while( -- ptr >= food)

puts(ptr);

}

1. Что напечатает нижеследующая программа?

main ()

{

static char goldwyn[28] = "аз я считываю"

static char samuel[40] = “Каждый p" ;

char \*quote = " часть строки.";

strcat(goldwyn, quote);

strcat(samuel, goldwyn);

puts(samuel);

}

1. Создайте функцию, которая использует указатель строки в качестве аргумента и возвращает указатель, ссылающийся на первый пробел в строке в указанном месте или после него. Если она не находит ни одного пробела, то пусть возвращает NULL-указатель.

**Ответы.**

1. Класс памяти должен быть extern или static; инициализация должна включать символ ' \0'.
2. До встречи в буфете.

о встречи в буфете.

До вст

вст

1. и

ми

мми

умми

Йумми

1. Каждый раз я считываю часть строки

char \*strblk(string)

char \*string;

{

while(\*string != ' ' && \*string != '\0')

string++ ; /\* останавливается на первом пробеле или нуль-символе \*/

if(\*string = = '\0')

return(NULL); /\* NULL = 0 \*/

else

return(string);

}

**Упражнения.**

1. Создайте функцию, которая считывает очередные n символов при вводе, включая символы пробелов, табуляции и новой строки.
2. Модифицируйте последнюю функцию таким образом, чтобы она останавливалась после ввода n символов или после первого символа пробела, табуляции или новой строки независимо от того, какой из них идет первым [только не используйте функцию scanf ()].
3. Создайте функцию, которая считывает очередное слово при вводе, определите слово как последовательность символов, не включающую символы пробела, табуляции или новой строки
4. Создайте функцию, которая ищет первое появление определенного символа в определенной строке. Функция должна возвращать указатель, ссылающийся на этот символ, в случае успешного поиска или NULL, если символ в строке не найден.