Министерство высшего и среднего специального образования России
Нижегородский Государственный Университет им Н.И. Лобачевского
Факультет Вычислительной Математики и Кибернетики
Кафедра ИИСГео

Язык программирования Си

Курс лекций для студентов д/о ВМК

Разработал: м.н.с. НИИ ПМК Васин Д.Ю.

Н.Новгород 2003

Содержание

Литература

1. **Керниган Б., Ритчи Д.** Язык программирования Си: Пер. с англ. М.: Финансы и статистика, 1992г. 271 с.
2. **Хенкок Л., Кригер М.** Введение в программирование на языке Си: Пер. с англ. М.: Радио и связь, 1986г. 191 с.
3. **Болски М.И.** Язык программирования Си: Пер. с англ. М.: Радио и связь, 1988г. 96 с.
4. **Джехани Н.** Программирование на языке Си: Пер. с англ. М.: Радио и связь, 1988г. 270 с.
5. **Уэйт М., Прата С., Мартин Д.** Язык Си: Руководство для начинающих. М.: Мир, 1988г. 512 с.
6. **Трой Д.** Программирование на языке Си для персонального компьютера IBM PC: Пер. с англ. М.: Радио и связь, 1991г. 432 с.
7. **Прокофьев Б.П., Сухарев Н.Н., Храмов Ю.Е**. Графические средства Borland C и Borland C++. М.: Финансы и статистика, СП «Ланит», 1992г. 160 с.
8. **Тондо К., Гимпел С.** Язык Си. Книга ответов: Пер. с англ. М.: Финансы и статистика, 1994г. 160 с.

1. Введение в язык Си

Язык Си был разработан как универсальный язык системного программирования. К его первым приложениям относится такое системное программное обеспечение, как операционные системы - ОС, компиляторы и редакторы. Он использовался и для таких приложений, как системы управления базами данных - СУБД, программы обработки крупноформатных бланков, научно-инженерные программы и программы обработки текстов. Он служит основным языком программирования для популярной ОС UNIX™, но используется и в других операционных средах.

1.1. История создания языка Си

Язык программирования Си был разработан в 1972 году в фирме Bell Laboratories (отделение известной телефонной компании AT&T) Деннисом Ритчи, одним из первых пользователей операционной системы Unix. Задумывался он не как универсальный алгоритмический язык, а, скорее, как инструмент для развития операционной системы и создания новых обслуживающих программ (утилит). Такой подход характерен для большинства системных программистов, разрабатывающих сложные проекты и придумывающих для облегчения своего труда различные сервисные процедуры, макрокоманды и т.п. По завершению разработки, как правило, эти инструментальные наборы предаются забвению или, в лучшем случае, остаются в личных архивах авторов. Язык Си эта участь миновала. Вполне возможно, что его становлению способствовало последующее всемирное признание операционной системы Unix. Его истоками можно считать язык BCPL (Basic Combined Programing Language - основной комбинированный язык программирования), разработанный Мартином Ричардсоном в Кембридже (Англия). В 1970 году специалисты по программированию, работавшие в фирме Bell Laboratories, разработали вариант языка BCPL, получивший название Би. Он использовался при разработке ранней версии ОС UNIX™ для компьютеров PDP-11™ фирмы Digital Equipment. В языке Би не было типов данных: его единственным объектом было машинное слово. Для получения доступа к отдельным машинным словам в нем использовались переменные, содержащие «указатели». Этот упрощенный взгляд на машину оказался неприемлемым в первую очередь потому, что в памяти компьютера PDP-11™ (прототип СМ-4) (как и в IBM PC) адресация осуществляется по байтам и наряду с целочисленными арифметическими операциями PDP-11™ выполняет операции над значениями с плавающей точкой. Но в языке Би отсутствовала адресация байтов, не различались целые значения и значения с плавающей точкой, а также не обеспечивались удобные способы выполнения арифметических операций над этими значениями.

Указанные недостатки вызвали преобразование языка Би в язык Си, основным отличием которого от Би было наличие типов данных. Каждое определение или объявление данных в языке Си задает тип этих данных, по которому можно установить, какой объем памяти требуется для хранения объекта и как интерпретировать его значения. Типы данных позволяют задавать байтовые символьные объекты, целые объекты и объекты с плавающей точкой. Язык Си унаследовал идею применения указателей и обеспечил возможность использования более сложных структур данных, таких как массивы и структуры.

Первым программным продуктом, написанным почти полностью на Си, был компилятор с языка Си в код машинных команд PDP-11/20 (прототип СМ-4). В 1973 г. Д.Ритчи и К.Томпсон переписали на Си большую часть операционной системы Unix.

В процессе перевода Unix из однопользовательской операционной системы, ориентированной на работу в конкретной ЭВМ, превратилась в мобильную операционную систему коллективного пользования. Успех этой операции, в значительной мере, предопределил популярность новой операционной системы и ее базового инструмента - языка Си. В 1976 г. Д.Ритчи и К.Томпсон перенесли Unix с ЭВМ фирмы DEC на компьютеры другой архитектуры (Interdata 8/32), практически ничего не изменив в ядре операционной системы, написанном на Си. Точно таким же образом Unix распространялся на десятках машин различных типов.

В 1978 г. появилась первая книга, посвященная описанию Си и технике программирования на этом языке, которая с большим запозданием была переведена на русский язык (Б. Керниган, Д. Ритчи, А. Фьюэр. Язык программирования Си. Задачи на языке Си. - М.: Финансы и статистика, 1985). От фамилий двух первых авторов произошло сокращенное обозначение первого, никем не утверждавшегося, но принятого всеми программистами стандарта языка Си - K&R.

Дальнейшая работа по совершенствованию языка Си и принятию в 1987г. первого настоящего стандарта ANSI C была выполнена на общественных началах рабочей группой при Американском Национальном Институте Стандартов. Возглавлял эту работу сотрудник Bell Labs Лэрри Рослер. Наиболее серьезный вклад в развитие языка Си за последние годы внес еще один представитель той же лаборатории Бьерн Страуструп, который ввел в обращение новые объекты - классы, объединяющие данные и обрабатывающие их функций. С 1983 г. за расширенной версией языка Си с классами закрепилось название C++.

Первые версии Си подвергались серьезной критике за отсутствие достаточной строгости, приводившей к многочисленным ошибкам из-за работы с неинициализированными переменными, отсутствия контроля за выходом индексов у элементов массивов за установленные пределы, несоответствия типов формальных и фактических параметров функций и т.п. Перед системными программистами Bell Labs эти проблемы остро не стояли, т.к. они пользовались специальной программой Lint, которая проводила тщательный анализ программ, написанных на Си, перед их трансляцией и выполнением. Для рядовых пользователей ситуация изменилась с появлением интегрированных сред, среди которых наибольшую популярность приобрели Турбо-системы фирмы Borland. Первая версия Borland C, работавшая в среде MS-DOS, была выпущена в 1987 г. В настоящее время фирма Borland вышла на рынок с версией 5.02, предназначенной для работы под управлением Windows. Известны и другие реализации языка Си на IBM–совместимых ПК - Microsoft C, Lattice C, Zortech C, Symantec C. Но в нашей стране продукция фирмы Borland получила наибольшее распространение.

1.2. Сравнение с другими языками программирования

Наряду с языком Си при работе на компьютере IBM PC можно пользоваться многими другими языками программирования, например языком ассемблера, языками BASIC, FORTRAN, COBOL, PL1 и PASCAL. Чтобы сравнить их с языком Си, необходимо уяснить цели их разработки.

АССЕМБЛЕР - язык программирования, ближе всего соответствующий системе команд микропроцессора. Хотя этот язык позволяет программисту полностью использовать возможности компьютера, он не является естественным языком для алгоритмических выражений. Программисты называют его «языком программирования нижнего уровня», поскольку он близок к машинному языку - языку программирования самого низкого уровня. Этот язык преобладал при разработке системного программного обеспечения, например ОС MS DOS полностью написана на языке ассемблера, а также при разработке компиляторов и загрузчиков, поскольку в подобных программах нередко требуется выполнять сложные манипуляции над данными.

Программисты-прикладники - студенты, ученые, инженеры - вскоре ощутили потребность в более естественных языках, которые бы упрощали программирование алгоритмов решения их задач. Для использования языков высокого уровня необходимо иметь компилятор (или интерпретатор), т.е. программу, которая преобразует операторы языка высокого уровня в машинный язык микропроцессора. Обсудим наиболее распространенные языки высокого уровня.

ФОРТРАН - представляет собой естественный язык для выражения математических алгоритмов. Пользователи этого языка рассматривают компьютер как мощный вычислитель. Данный язык был разработан в 50-х годах, а затем совершенствовался и несколько раз расширялся. Фортран чрезвычайно популярен среди ученых и инженеров, однако он совершенно не пригоден для разработки системного программного обеспечения.

Т.к. Фортран предназначен в основном для вычислений, то в нем отсутствуют развитые средства для представления структур данных и невозможен прямой доступ к ячейкам памяти. Он не обеспечивает выполнение некоторых операций, имеющихся в языке ассемблера (операций над битами), не дает возможности пользоваться составными данными (за исключением массивов) и применять некоторые современные методы программирования, например рекурсивное программирование. На Фортране можно писать большие программы, разбивая задачу на части (называемые подпрограммами или функциями), которые программируются независимо, а затем объединяются в единое целое.

КОБОЛ - преобладающий язык в задачах обработки административных и финансово-экономических данных. Пользователи Кобола рассматривают компьютер как средство обработки финансово-экономических данных. В язык включены развитые средства выполнения операций ввода-вывода. Кобол позволяет создавать большие программы за счет использования независимо создаваемых и компилируемых подпрограмм. Однако, как и в Фортране, в нем отсутствуют многие современные средства программирования, например реккурсивное программирование и адресация ячеек памяти. Равным образом Кобол не пригоден для системных приложений.

БЕЙСИК - был разработан как упрощенная версия языка Фортран и предназначался для обучения начинающих программистов. Пользователи Бейсика рассматривают компьютер как программируемый калькулятор. В настоящее время он один из наиболее популярных языков среди всех языков, используемых на ПК.

Большинство реализаций Бейсика являются интерпретаторами, а не компиляторами. Интерпретатор представляет собой программу, которая непосредственно исполняет предложения языка, а не преобразует их в программу на машинном языке, как это делает компилятор. Бейсик хорош для написания небольших диалоговых программ. Среда программирования, обеспечиваемая интерпретатором, позволяет быстрый ввод и проверку программ. Интерпретатор может оказаться неудобным для более серьезного программиста, разрабатывающего программный продукт, поскольку скорость исполнения интерпретируемой программы обычно ниже скорости исполнения откомпилированной программы.

В Бейсике отсутствуют настоящие подпрограммы, имеющиеся в Фортране и Коболе; поэтому программы на языке Бейсик быстро становятся большими и неудобными для дальнейшего развития. У него отсутствуют операторы, требуемые для выполнения прямого доступа к ячейкам памяти, бит-ориентированные и некоторые другие операции, необходимые при системном программировании. В Бейсике также отсутствует возможность определения структур данных, отличных от массивов и строк символов.

ПЛ1 - был разработан фирмой IBM для замены и Фортрана, и Кобола. Он обладает вычислительными возможностями Фортрана и средствами Кобола по обработке файлов данных. Он обеспечивает много дополнительных возможностей, например операции над битами, прямой доступ к ячейкам памяти, и рекурсивное программирование. Однако ПЛ1 не оправдал ожиданий своих создателей и не вытеснил ни Фортран, ни Кобол. Возможно, это вызвано тем, что средств языка ПЛ1 столько, что его трудно выучить, а компиляторы представляют из себя очень большие программы и пользоваться ими можно только на больших машинах. Это мешает использованию ПЛ1 на персональном компьютере.

Паскаль - был разработан как язык для преподавания современных принципов программирования. Пользователи Паскаля рассматривают компьютер как очень структурированную машину, способную манипулировать многими различными типами данных. Этот язык полезен для решения прикладных задач различных типов. Он обеспечивает применение современных методов программирования и хорош как средство структурного программирования. Как и другие языки программирования высокого уровня, большинство вариантов реализаций Паскаля допускает раздельную разработку подпрограмм, что необходимо при создании больших систем. Операторы ввода-вывода также являются операторами языка.

Будучи ориентированным на обучение, Паскаль строго типизирован. Это означает, что каждая переменная программы должна быть описана как содержащая определенный тип данных, и язык требует строгого соответствия типов переменной и присваиваемого ей значения. Таким образом, пользователи Паскаля рассматривают компьютер как современную универсальную машину, которая должна подчиняться структурированной системе правил. Паскаль получил широкое распространение в промышленных приложениях, но наиболее интенсивно используется все же в учебных целях.

Си - при разработке этого языка был принят компромисс между низким уровнем языка ассемблера и высоким уровнем описанных выше языков. Пользователи языка Си рассматривают компьютер в ракурсе, объединяющем точки зрения пользователей языков ассемблера и Паскаль. В языке Си предусмотрено много операций, непосредственно исполняемых большинством микропроцессоров (например, прямой доступ к ячейкам памяти и манипулирование битами); в то же время он дает программисту возможность выражать свои алгоритмы и данные наиболее подходящими средствами, использующими такие традиционные конструкции языков программирования высокого уровня, как итерация, выбор и принятие решения. Язык Си обеспечивает возможности структурирования данных, отсутствующие в языке ассемблера, но присущие современным языкам программирования высокого уровня. Он позволяет разрабатывать большие, но структурированные программы, предоставляя возможность отдельной разработки подпрограмм (в отличие от Бейсика и некоторых вариантов реализации языка Паскаль).

В отличие от ПЛ1 язык Си компактен. Его создатели ориентировались на мини-компьютеры, и разработанный ими компилятор занимал всего лишь 12 Кб ОП. Размер компилятора языка Си делает его идеальным для применения на персональном компьютере. Чтобы сохранить язык Си компактным, его создатели удержались от соблазна включить в него множество операций, отсутствующих у большинства микропроцессоров. Так, в языке Си нет встроенных операций для манипулирования строками и даже встроенных операций ввода-вывода. Эти возможности, меняющиеся в зависимости от компьютера или приложений, были вынесены из собственно языка и реализованы как подпрограммы, которые могут быть вызваны из программы, написанной на языке Си.

Другой целью создателей языка Си была разработка мобильного языка, который можно было бы использовать для разработки системного программного обеспечения. Язык программирования называют мобильным, если написанные на нем программы могут быть без труда перенесены из одной вычислительной среды в другую. Системное программное обеспечение, написанное на языке ассемблера, не может быть мобильным, поскольку языки ассемблера разных компьютеров различаются (особенно если эти компьютеры выпущены разными фирмами-производителями). Напротив, программы, написанные на языке программирования высокого уровня, мобильны, поскольку язык должен быть одним и тем же независимо от того, на каком компьютере и в какой ОС он используется. В прошлом прикладное программное обеспечение - ПО было более мобильным, чем системное ПО, поскольку на новом компьютере прикладную программу можно было просто заново откомпилировать. Авторы Си заполнили этот промежуток, создав компилируемый язык, пригодный для разработки системного ПО. Так как многие конструкции высокого уровня (ввод-вывод, манипулирование структурами данных) реализованы вне языка Си как подпрограммы, то при необходимости их можно написать ориентированными на конкретное приложение или конкретную ОС, не затрагивая при этом компилятор языка Си или любое ПО, написанное на этом языке. Можно приобрести библиотеки наиболее широко используемых подпрограмм; в

действительности библиотека процедур ввода-вывода и других подпрограмм, называемая *стандартной библиотекой*, поставляется большинством производителей компиляторов. Наконец, дополнительное свойство языка Си, называемое условной компиляцией, позволяет программисту изолировать машинно-зависимые операторы и контролировать их компиляцию в другой среде. Это дополнительно повышает мобильность ПО, написанного на языке Си.

Таким образом, в плане структур данных и управления точка зрения пользователей языка Си на компьютер на уровень выше, чем у пользователей языка ассемблера. Тем не менее он ориентирован на приложения конкретного типа не в такой степени, как большинство других языков программирования высокого уровня. Чтобы писать программы на языке Си, программист должен обладать достаточно высокой квалификацией: компиляторы языка Си не контролируют согласование типов данных в отличие от компиляторов языка Паскаль. В результате Си является более гибким языком, но при программировании на нем легче ошибиться.

1.3. Пользование компилятором

Компилятор представляет собой системную программу, которая преобразует некоторый «язык высокого уровня» в язык компьютера - язык ассемблера или машинный язык. Интерпретатору Бейсика этого не требуется, поскольку он непосредственно исполняет каждый оператор. Интерпретаторы хороши при написании небольших программ, не требующих многократного исполнения. Компиляторы более удобны для разработки больших программ, которые должны исполняться много раз. Это обусловлено тем, что интерпретация программы осуществляется намного медленнее исполнения на компьютере ранее откомпилированной программы.

По сравнению с интерпретатором при работе с компилятором требуется выполнить несколько дополнительных операций (рис.1).

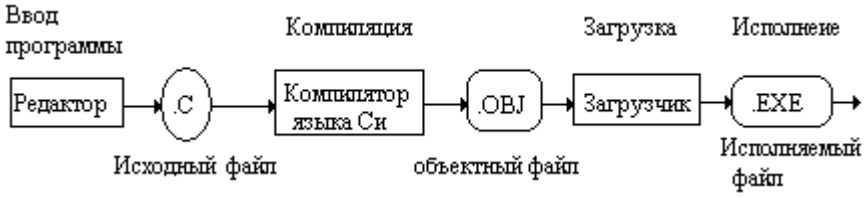


Рисунок 1

Программа готовится с помощью текстового редактора или программы обработки текста. Она запоминается в *исходном файле*, который преобразуется компилятором в *объектный файл.* Затем объектный файл, возможно, вместе с другими объектными файлами, полученными ранее (может быть с помощью других компиляторов), преобразуется в *исполняемый файл* программой, называемой *загрузчиком* или *редактором связей*. Этот файл может быть исполнен непосредственно компьютером.

1.4. Внутренняя структура программы на языке Си для IBM PC (альтернативные модели распределения памяти)

Исполняемая программа на языке Си состоит из четырех частей: *областей команд, стека, статических и динамических данных.* Область команд содержит машинные команды. Стек используется для временного хранения данных и адресов возврата при вызовах подпрограмм. Область статических данных обеспечивает хранение значений переменных программы, а в области динамических данных размещаются уже при исполнении программы дополнительные данные, которые могут понадобиться в процессе ее работы.

Современные компиляторы языка Си обеспечивают реализацию нескольких моделей распределения памяти. Под областью данных в табл.1 подразумевается область динамических данных модели распределения памяти, используемой в программе на языке Си. Размер области статических данных во всех моделях обычно не превышает 64 Кб, аналогично размер области стека также не превышает 64 Кб.

Таблица 1 Альтернативные модели распределения памяти

Модель распределения памяти	Максимальный размер области команд, байт	Максимальный размер области данных, байт
Малая модель	64К	64К
Модель больших кодов	До 1М	64К
Модель больших данных	64К	До 1М
Большая модель	До 1М	До 1М

Модель распределения памяти задается на стадии компиляции путем указания соответствующих параметров при вызове компилятора языка Си. Каждое значение параметра, задающего модель

распределения памяти, заставляет компилятор генерировать несколько иные машинные коды. Для большинства компиляторов это влечет за собой требование, чтобы при раздельной компиляции программ на языке Си задавалась одна и та же модель памяти.

Выбор модели зависит от требуемого приложения. Рекомендуется выбирать наименьшую возможную модель, которая удовлетворяет Вашим требованиям, поскольку применение больших моделей влечет за собой увеличение размера области команд и замедление работы программы.

2. Интегрированная среда Borland C

2.1 Основные компоненты интегрированной среды Borland C

Для создания ваших программ вы можете использовать любой текстовый редактор ASCII. Затем можно использовать компилятор командной строки (файл TCC.exe) для компиляции и последующего запуска на выполнение программ из командной строки DOS. Все же такой путь может показаться неудобным, и вы можете использовать интегрированную среду разработки Borland C (файл TC.exe).

Интегрированная среда Borland C - это более чем просто быстрый Си-компилятор. Когда Вы запускаете программу Borland C, то весь комплекс инструментальных средств, необходимых Вам для написания, редактирования, компиляции, компоновки и отладки Ваших программ, оказывается у Вас под рукой. Весь этот комплекс возможностей заключен в Интегрированной Среде Разработки.

Кроме того, среда разработки программ системы Borland C предоставляет следующие дополнительные возможности, которые еще больше упрощают процесс написания программ:

- ✓ Возможность присутствия на экране монитора значительного числа окон, которые можно перемещать по экрану и размеры которых можно изменять;
- ✓ Наличие поддержки мыши;
- ✓ Наличие блоков диалога;
- ✓ Наличие команд отсечения и вклейки (при этом допускается копирование из окна Help (Подсказка) и между окнами редактора);
- ✓ Возможность быстрого перехода к другим программам, например, к программе TASM - Турбо Ассемблеру и обратного возврата;
- ✓ Наличие в редакторе макроязыка.

2.2 Загрузка интегрированной среды Borland C

В простейшем случае для запуска интегрированной среды необходимо перейти в ту директорию, в которой находится система Borland C (по умолчанию - это \TC\BIN), и загрузить файл TC.exe, набрав в командной строке операционной системы TC и нажав клавишу <Enter>.

При необходимости вместе с командой TC можно использовать один или несколько параметров, определяющих режимы работы среды.

2.3. Основной экран Borland C

Интегрированная Среда Разработки насчитывает в себе три визуальных компоненты: строку меню у верхнего края экрана, оконную область в средней части экрана и строку состояния у нижнего края экрана. В результате выбора некоторых элементов меню на экран будут выдаваться блоки диалога.

2.4. Выход из системы Borland C

Выход из системы Borland C можно осуществить тремя способами:

1. **Первый способ.** Окончательный выход из системы в среду DOS. При использовании этого способа для возврата в систему Borland C необходимо вновь набирать на клавиатуре tc. Реализуется данный способ путем выполнения команды File | Quit (Файл | Выход). Для выбора данной команды необходимо нажать клавишу <F10>, выбрать световым указателем позицию File, нажать <Enter> и в выпавшем меню выбрать Quit. Можно также нажать комбинацию клавиш <Alt X>. Если вы внесли в тексты программ какие-либо изменения, которые еще не были сохранены, то система Borland C выдаст сообщение, в котором спрашивается, хотите ли Вы сохранить Ваши программы перед выходом из системы.
2. **Второй способ.** Временный выход из системы Borland C для того, чтобы ввести какие-либо команды в ответ на запрос DOS. Необходимо выбрать команду File | DOS Shell (Файл | Оболочка DOS). Система Borland C по прежнему останется в памяти, но Вы при этом оказываетесь в DOS. После выхода из системы Borland C Вы можете ввести любую обычную команду DOS и даже можете запустить из командной строки какие-либо другие программы. Когда Вы готовы вернуться в систему Borland C, введите в командной строке команду EXIT

(выход) и нажмите клавишу <Enter>. Вид системы Borland C будет в точности соответствовать тому, который был при выходе из нее.

3. **Третий способ.** Временный выход из системы Borland C для того, чтобы временно перейти к другой программе, не покидая окончательно систему Borland C. Для этого выбираем из меню «Системное» - обозначается тремя горизонтальными черточками и находится слева в строке меню. Если в это меню не были инсталлированы никакие программы, то они могут быть добавлены с помощью команды Options | Transfer (Параметры | Переход).

2.5. Получение помощи

Borland C, как и другие программные продукты фирмы Borland, дает контекстно-зависимую от экрана справочную информацию. При нажатии одной единственной клавиши Вы можете получить справочную информацию в любой точке любого меню Borland C.

Система подсказки содержит информацию практически по всем аспектам интегрированной среды и системы Borland C, любому зарезервированному слову Borland C или библиотечной функции. Кроме того можно скопировать из окна подсказок примеры программ в программу пользователя.

Borland C имеет также «систему минимальных подсказок», которая всегда видна на экране. Строка состояния в нижней части экрана всегда содержит все активные клавиши или команды доступные для текущего окна, меню или блока диалога и дает краткое описание назначения текущего управляющего элемента меню или блока диалога.

Для открытия окна Help (Подсказка) следует осуществить одно из следующих действий:

- в произвольный момент времени нажать клавишу <F1> (в том числе, когда вы находитесь в блоке диалога или когда выбрана какая-либо команда меню).
- нажать комбинацию клавиш <Ctrl F1> для получения подсказки по языку, когда активным является окно редактирования и курсор позиционирован на каком-либо слове.
- подвести указатель мыши к слову Help (Подсказка), когда оно появляется в строке состояния или в блоке диалога, и нажать кнопку мыши.

Экранный кадр подсказки часто содержит ключевые слова (выделенный световым атрибутом текст), который вы можете выбрать, чтобы получить дополнительную информацию. Нажатие клавиши <Tab> приводит к перемещению к любому из ключевых слов; после этого для получения дополнительной информации необходимо нажать клавишу <Enter>. При работе с мышью для перехода к тексту подсказки, на который ссылается данный элемент, можно подвести указатель мыши к этому слову и дважды подряд нажать левую кнопку мыши.

Работая с редактором, у вас может возникнуть необходимость получить описание на различные библиотечные функции. Если Вы захотите получить информацию о функции (например, printf), переместите курсор на экране под имя этой функции, нажмите <Ctrl F1>, и вы получите желаемое описание.

Чтобы выйти из HELP и вернуться к меню, с которым Вы работали перед тем, как выдать запрос на получение справочной информации, нажмите <Esc>.

2.6. Редактор Интегрированной среды

2.6.1. Основные характеристики редактора интегрированной среды.

Редактор интегрированной среды содержит следующие средства:

- поддержка мыши;
- поддержка больших по размерам файлов (размеры файлов могут превышать 64 Кб; общее ограничение для всех файлов в редакторе в совокупности равняется 8 Мб);
- использование клавиши <Shift> в сочетании с четырьмя клавишами управления курсором (вверх, вниз, влево, вправо) для выделения фрагментов текста;
- окна редактирования, которые можно перемещать и перекрывать друг с другом, и размеры которых можно менять;
- возможности работы с несколькими файлами, что позволяет вам открывать несколько файлов одновременно;
- мультиоконная система, которая позволяет вам просматривать в нескольких окнах один и тот же файл или разные файлы;
- мощный макроязык редактора, при помощи которого вы можете создавать свои собственные команды редактора;
- возможности вклейки текста или примеров программ из окна Help (Подсказка);
- текстовый буфер (Clipboard), содержимое которого можно редактировать; этот текстовый буфер позволяет отсекать, копировать текст между окнами и вклеивать его в текст окна;

- функция Transfer (Перенос), которая позволяет вам запускать другие программы и захватывать выводимые ими данные в редактор, не покидая среду Borland C

2.7. Основы работы в среде Borland C

2.7.1. Запуск интегрированной среды, создание и сохранение файлов

Для упрощения запуска интегрированной среды рекомендуется установить путь в директорию BIN в команде path файла AUTOEXEC.BAT. Тогда для запуска интегрированной среды достаточно ввести команду bc. Появится основной экран Borland C, в котором будет открыто 1 окно редактора. Файл в этом окне по умолчанию имеет имя NONAME00.C.

Введем текст классической программы для начинающих на языке Си

```
#include <stdio.h>
void main(void)
{
   printf (“Здравствуйте !!!\n”);
}
```

Чтобы сохранить файл на диске выполните команду Save as из меню File. В открывшемся окне введите имя файла Hello.c и нажмите <Enter>. Текст программы будет сохранен в файле Hello.c в текущей директории.

2.7.2. Компилирование и запуск программы на выполнение

При создании программы исходный файл сначала компилируется в объектный файл (файл в машинных кодах с расширением .OBJ). Затем объектный файл посылается на компоновку, чтобы он был преобразован в выполняемый файл с расширением .EXE. Компоновщик копирует в ваш объектный файл необходимые подпрограммы из стандартных библиотечных файлов.

Самый легкий путь для создания выполняемых программ в среде Turbo C++ - это нажать клавиши F10, а затем клавишу C, чтобы войти в меню Compile (или нажать <Alt+C>). Затем выбрать пункт MakeEXEFile (клавиша <F9> - «горячая» клавиша для создания .EXE файла). Заметим, что меню Compile сообщает вам имя объектного (с расширением .OBJ) файла, который будет откомпилирован в файл .EXE.

На экране появится окно компиляции. Если все произойдет успешно, в окне компиляции появится мигающее сообщение:

Success: Press any key (Успех: нажмите любую клавишу).

Примечание: В случае ошибки в вашей программе вы увидите сообщения об ошибках или предупреждениях в окне сообщений в нижней части экрана. Если так случится, убедитесь, что ваша программа выглядит именно так, как было описано выше, затем откомпилируйте ее снова.

Если появится сообщение об ошибке, говорящее о том, что Borland C не может найти включаемые файлы (файлы с расширением .h), то скорее всего Borland C не был установлен с подкаталогами по умолчанию. Это делается с помощью команды Options | Directories. Но об этом несколько позднее.

Для запуска программы выберите пункт Run или нажмите клавиши <Ctrl+F9>, которые являются клавишами быстрого вмешательства для запуска программы.

Вы увидите как мигнет экран, и затем вы снова вернетесь в основной экран Borland C. Для просмотра на экране сообщений программы, выберите Run|UserScreen или нажмите <Alt+F5>. Это вызовет появление экрана пользователя, на который наша программа выводила сообщения.

Пользовательский экран должен содержать сообщение:

Здравствуйте !!!

После того, как вы проверите сообщения программы, нажмите любую клавишу для возврата в экран Borland C.

2.7.3. Закрытие Окна Редактирования

Работа с файлом Hello.c закончена. Для закрытия окна редактирования надо выбрать Window|Close (или нажать <Alt+F3>).

При работе с мышью можно выбрать закрытый квадратик в левом верхнем углу экрана для закрытия окна редактирования.

2.7.4. Выход из Borland C

После окончания работы с файлом нужно сделать две вещи:

- записать внесенные изменения в файл на диск;
- выйти из интегрированной среды Borland C и перейти в DOS.

Файл уже был записан, поэтому последний шаг состоит в выходе из Borland C и возврате в DOS. Для этого нужно выбрать команду File|Quit или нажать <Alt+X>.

2.7.5. Какие файлы создаются в процессе трансляции и компоновки

Посмотрим какие файлы мы создали. Находясь в текущей директории, мы увидим список следующих файлов:

Имя файла	Длина	Дата создания/модернизации	Время создания/модернизации
Hello C	XXXX	ЧЧ-ММ-ГГ	ЧЧ:ММ
Hello OBJ	XXXX	ЧЧ-ММ-ГГ	ЧЧ:ММ
Hello EXE	XXXX	ЧЧ-ММ-ГГ	ЧЧ:ММ

Первый файл - Hello.c - является исходным текстом вашей программы.

Второй файл - Hello.obj - является объектным файлом. Он содержит двоичные машинные инструкции (объектные коды), полученные с помощью компилятора Borland C.

Последний файл - Hello.exe - является загрузочным модулем, сделанным компоновщиком Borland C. Он содержит не только код файла Hello.obj, но и все необходимые подпрограммы (такие как printf), которые компоновщик поместил в него из библиотечного файла. Для запуска любого выполняемого файла из DOS вам необходимо ввести его имя без расширения .EXE.

Для запуска Hello.exe из DOS введите hello и нажмите клавишу <Enter>. На экране появится сообщение Здравствуйте !!! и затем снова приглашение DOS.

2.7.6. Загрузка в редактор и редактирование вашей программы

В данном разделе продемонстрируем выполнение следующих задач с использованием средств интегрированной среды Borland C :

- открытие готового файла в окне редактирования;
- копирование файла в другое окно редактирования;
- использование редактора для модификации скопированного файла;
- запись измененного файла на диск.

ЗАГРУЗКА ФАЙЛА В ОКНО РЕДАКТОРА

Для запуска Borland C снова перейдем в созданную нами рабочую директорию и введем команду DOS bc. Поскольку последний файл, с которым мы работали в интегрированной среде был файл Hello.c, то он автоматически будет загружен в первое окно редактора.

В противном случае, для загрузки программы Hello.c в окно редактора интегрированной среды Borland C нужно выбрать команду File|Open или нажать клавишу <F3>. В любом случае на экране должен появиться диалоговый блок Загрузки Файла.

Заметим, что по умолчанию в блоке диалога выбрана клавиша Open (Открыть). Если нужно выбрать Replace (Заменить) вместо Open, то файл заменит тот файл, который находится в данный момент в текущем окне редактирования, вместо помещения его в новое окно. Оставим выбранной клавишу Open.

В данном блоке диалога есть два способа выбрать файл для его открытия:

- ввести имя файла в блок ввода имени, или
- указать файл из списка файлов.

Воспользуемся списком файлов:

1. Нажать <Tab> для активации списка файлов.
2. Обычно для подсвечивания нужного пользователю файла используются клавиши со стрелками. В данном конкретном случае файл Hello.c уже подсвечен, т.к. он стоит в списке первым.
3. Нажать <Enter>.

Если используется мышь, то нужно сделать двойное нажатие, указав на имя файла в списке.

После нажатия на клавишу <Enter> содержимое файла Hello.c появится в окне редактирования. Обратите внимание на номер окна 1 в правом верхнем углу окна редактирования. В левом нижнем углу видна информация о текущем номере строки и колонки окна редактирования.

Существует возможность загрузки интегрированной среды и исходного файла из командной строки, что делает ненужным выполнение предыдущих шагов. Интегрированная среда допускает аргумент в командной строке, который означает имя исходного файла, который должен быть загружен в редактор. Таким образом, команда

bc hello

поместит файл hello.c в редактор.

СОЗДАНИЕ НОВОГО ФАЙЛА

Особенностью интегрированной среды Borland C является многооконность - в каждый момент времени можно иметь более одного открытого окна редактирования. Можно открыть разные или один и тот же файл в каждом окне, обмениваться информацией между окнами, делая вырезки и вставки, и легко переходить из одного окна в другое.

Сейчас файл hello.c открыт в окне редактирования и можно начать его редактирование. Но было бы нецелесообразно менять текст оригинального файла, откроем новое окно редактирования с помощью File| New.

Файл в новом окне (окно 2) имеет имя NONAME00.C, позднее назовем его SUM.C. Двойная рамка вокруг нового окна редактирования указывает на то, что это активное окно.

Для перехода к конкретному окну редактирования нужно нажать <Alt> и номер окна.

Вернемся в первое окно редактирования, нажав <Alt+1>. Нужно скопировать содержимое файла hello.c в NONAME00.C.

ВЫБОР БЛОКА

Отметку блока можно делать с клавиатуры и мышью:

- На клавиатуре нужно подвести курсор к первой или последней букве блока, нажать клавишу <Shift> и, не отпуская ее, нажимать клавиши со стрелками (или <Home>,<End>,<PgDn>,<PgUp>) до выделения всего нужного блока.
- Мышью нужно указать первый символ блока и, не отпуская кнопку мыши переместить ее в конец блока.

Для отмены выделения блока нужно нажать кнопку на мыши в произвольном месте или нажать комбинацию клавиш <Ctrl+K+H>.

Выбранный блок будет показан инверсным цветом (подсвечен). В качестве блока нужно выбрать весь текст hello.c.

КОПИРОВАНИЕ И ВСТАВКА

Мы выбрали весь текст в окне hello.c. Выберем Edit|Copy (или нажмем <Ctrl+Ins>). Данная операция скопирует выбранный текст в специальное место памяти, называемое буфером. Текст, находящийся в буфере, можно скопировать в новое место этого же окна или в другое окно.

Вернемся к NONAME00.C: нажмем <Alt> и номер окна (или выберем окно редактирования с помощью мыши). Если нужно перейти к конкретному окну, а номер его неизвестен, то следует воспользоваться Window|List.

Для вставки текста из буфера в окно редактирования, нужно выбрать Edit|Paste (или нажать <Shift+Ins>). Отменим теперь выделение блока, нажав кнопку мыши или клавиши <Ctrl+K+H>.

Теперь можно внести в текст некоторые изменения и записать файл NONAME00.C на диск, дав ему другое имя.

ВНЕСЕНИЕ ИЗМЕНЕНИЙ В ФАЙЛ

Внеся изменения в программу hello.c, создадим программу sum.c, которая позволяет ввести два целых числа и вывести их сумму на экран. При этом продемонстрируем возможности редактора по поиску и замене для изменения имени переменной.

Для ввода текста передвиньте курсор в необходимое место и наберите текст. Вы можете удалить строку текста, нажав <Ctrl+Y>, а также вставить строку, нажав <Ctrl+N>. Убедитесь, что вы работаете в режиме Insert (Вставка).

Отредактируем программу следующим образом:

```
#include <stdio.h>
void main(void)
{
   int a,b,i;
   printf("\n Введите два числа:"); scanf("%d%d",&a,&b);
   i=a+b;
   printf("\n %4d + %4d = %4d",a,b,i);
}
```

ПОИСК И ЗАМЕНА

Заметим, что имя переменной s не является очень информативным идентификатором. Заменим его на sum, так как это имя больше говорит о назначении переменной.

Замена однобуквенной переменной достаточно более коварная операция, чем это может показаться на первый взгляд. Нужно заменить все появления имени переменной i в тексте. Если пропустить хотя бы

одно, то программа не будет компилироваться. Но в то же время нельзя заменять все появления буквы i в тексте (например i в слове printf), а только имена переменной i.

К счастью, меню Search (Поиск) включает опцию выборочного поиска и замены. Будет использован диалог Замены.

Блок диалога Замены содержит два блока ввода, три набора селективных кнопок, набор блоков проверки и четыре кнопки действий. Мы собираемся заменить все появления имени переменной i в файле NONAME.C на sum, поэтому перейдем к началу файла (нажатие <Ctrl+PgUp> или с помощью мыши). Далее:

1. Выберем Search/Replace для открытия блока диалога Замены. При первом открытии блока диалога активным является блок ввода текста, подлежащего замене.
2. Введем i, что соответствует имени переменной, подлежащему замене, и нажмем кнопку <Tab> для активизации блока ввода нового текста.
3. Введем sum в качестве имени, которое заменит i. Нажмем <Tab> для передачи управления блоку Options режимов замены.

Режим чувствительности к регистрам букв и другие параметры уже выставлены так, как нам нужно. Но следует выставить режим проверки только целых слов Whole words, чтобы поиск не прерывался на каждой букве i в тексте программы. С помощью клавиши "↓" выберите эту позицию и нажмите на клавишу <Пробел>. После внесения этой установки (появится крестик) нажмите клавишу <Tab> или кнопку мыши для перехода на селективные клавиши Directions (направления).

4. Мы желаем осуществлять поиск сверху вниз, этот режим уже выставлен, поэтому с помощью клавиши табуляции перейдем на Scope (диапазон).
5. В этом блоке уже установлен Глобальный режим, что означает поиск во всем тексте. Нажав <Tab>, перейдем на Origin (точка отсчета).
6. В Origin нажмем клавишу со стрелкой вниз, что означает организацию поиска по всему тексту Entire scope, или укажем этот режим с помощью мыши.
7. Нажав клавишу <Tab> перейдем на клавишу Change All (замена всех) и нажатием <Enter> для инициализации операции поиска и замены (или выберем эту клавишу с помощью мыши).

При каждом обнаружении буквы i у пользователя будет запрашиваться подтверждение на замену. Нажать Y (да), если была обнаружена переменная i, и N (нет) в противоположном случае.

Нами был внесен ряд изменений в программу и теперь нужно записать этот измененный файл на диск. Выберем File | Save (или нажать <F2>), в результате чего на экране появится блок диалога Записи Файла Редактора. В блоке ввода введем sum.c и нажмем <Enter>.

ВСТАВКА ИЗ ОКНА ПОДСКАЗОК

В интегрированной среде Borland C имеется удобная возможность копирования текста примера программ из окна подсказки Help в окно редактора.

После копирования в окно редактора можно удалить ненужный текст, оставив то, что вам необходимо для использования в текстах своих программ.

Для выполнения этого действия необходимо выполнить следующее:

1. Выбрать Index из меню Help
2. Ввести название той функции, текст примера использования которой вы хотите скопировать (например, printf). Нажать <Enter> для вызова экрана-подсказки.
3. Пример уже выделен в качестве блока, поэтому для копирования его в буфер следует выбрать <Edit/Copy Example>
4. Нажать <Esc> для выхода из окна Подсказок Help.
5. Перевести курсор в то место файла, куда вы хотите вставить копируемый текст.
6. Выбрать <Edit/Paste> для помещения блока текста в файл.

После того, как пример скопирован в программу, вы можете изменить некоторые строки и удалить старые.

2.8 Работа с несколькими исходными файлами. Файлы проекта

2.8.1. Файлы проектов

НАЗНАЧЕНИЕ ФАЙЛОВ ПРОЕКТА

Ранее мы рассматривали только один исходный файл, поэтому можно было использовать команду <Compile | Make> для создания выполняемой программы без определения файла проекта. Однако при построении программы из нескольких исходных файлов Си необходимо точно сообщить Borland C, какие файлы используются, т.е. создать файл проекта.

В системе Borland C интегрированная среда заносит всю необходимую для построения программы информацию в файл проекта, имеющий двоичный формат.

В файл проекта входит следующая информация:

- имена всех файлов, входящих в проект;
- где их следует искать на диске;
- какие файлы зависят от других файлов (автоматически отслеживаемые зависимости);
- какие компиляторы и параметры командной строки должны использоваться при создании каждой из частей программы;
- куда следует поместить результирующую программу;
- размер кода, размер данных и число строк, полученных в результате последней компиляции.

ЗАГРУЗКА ФАЙЛОВ ПРОЕКТА

1. При запуске системы Borland C имя файла проекта с расширением *.prj можно задать в командной строке
   bc mypro.prj
2. Если в текущей директории находится только один файл проекта с расширением *.prj, интегрированная среда предполагает, что эта директория предназначена для данного проекта и загружает его автоматически. Таким образом, если текущая директория содержит только один файл проекта, и вы вводите в командной строке только команду bc, этот файл проекта будет загружен в память.
3. Находясь внутри и нтегрированной среды вы можете загрузить файл проекта посредством команды <Project | Open Project (Проект | Открыть проект)>.

Замечание. Когда файл проекта загружается из директории, не являющейся текущей, текущей директорией DOS делается та директория, из которой загружается проект.

ФАЙЛЫ МАКЕТА ЭКРАНА

С каждым проектом связан файл макета экрана. Этот файл имеет имя - <имя проекта.dsk>. Этот файл содержит статусную информацию по текущему проекту. Хотя ни одна часть содержащейся в нем информации не требуется для построения проекта, вся эта информация имеет к проекту непосредственное отношение.

Файл макета экрана включает в себя:

- контекстную информацию по каждому файлу проекта (т.е. позиция в файле, позиция окна на экране и т.д.);
- список «предыстории» для различных блоков ввода (например, образцов строк для поиска, масок файлов и т.д.);
- схему расположения окон на макете экрана.

СМЕНА ФАЙЛОВ ПРОЕКТА

Поскольку каждому файлу проекта соответствует свой собственный файл макета экрана, смена на другой файл проекта приводит к сохранению текущего файла макета экрана и использованию макета экрана нового проекта. Таким образом, смена существующего проекта на другой существующий проект может привести к смене всего макета экрана. Когда вы создаете новый проект (путем использования команды меню <Project | Open Project (Проект | Открыть проект)> и ввода имени нового файла с расширением *.prj), макет экрана нового проекта будет наследовать параметры макета предыдущего проекта. Когда вы выбираете команду <Project | Close Project (Проект | Закрыть проект), загружается стандартный файл проекта, и вы приходите к стандартному макету экрана.

2.8.2. Использование менеджера проекта

Поскольку большая часть программ состоит из нескольких файлов, желательно иметь возможность автоматической идентификации тех файлов, которые должны быть рекомпилированы и рекомпанованы. Встроенный менеджер проектов системы Borland C выполняет не только эти обязанности, но и многие другие.

Менеджер проектов позволяет вам задавать те файлы, которые относятся к описываемому проекту. Когда вы осуществляете рекомпиляцию проекта, менеджер проектов автоматически обновляет информацию, которая хранится в файле проекта.

Использование менеджера проектов не представляет затруднений. Для построения проекта следует:

- выбрать имя файла проекта (команда Project | Open Project (Проект | Открыть проект));
- добавить к проекту имена исходных файлов (команда Project | Add Item (Проект | Добавить элемент));
- указать системе Borland C++ скомпилировать и скомпоновать файлы, включенные в проект (команда Compile | Make EXE (Компилировать | Создать выполняемый файл)).

Затем, когда в меню Project (Проект) станут доступны команды организации проекта, вы можете:

- добавить имена файлов в проект или удалить их из него;
- задать параметры для обработки файла, внесенного в проект;
- просмотреть содержимое включаемых файлов для конкретного файла в проекте.

Рассмотрим на примере, как работает менеджер проектов.

Пусть у вас имеется программа, которая состоит из основного исходного файла с именем MYMAIN.C, дополнительного файла MYFUNCS.C, содержащего функции и данные, обращения к которым имеются в основном файле, и файла MYFUNCS.H.

Эти файлы составляют программу, которая будет описана для менеджера проектов.

Замечание: Указываемые далее имена могут быть теми же самыми, но могут и отличаться (это условие не касается расширений имен). Имя получаемого выполняемого файла (и любого файла, который создается компановщиком) основывается на имени файла проекта.

Первый этап заключается в том, чтобы указать системе Borland C++ имя файла проекта, который Вы собираетесь использовать. В данном случае мы назовем его MYPROG.PRJ. Заметьте, что имя файла проекта не совпадает с именем основного файла MYMAIN.C. Кроме того, именем выполняемого файла будет MYPROG.EXE.

Для того, чтобы перейти к меню Project (Проект), необходимо нажать комбинацию клавиш <Alt+P>. Затем следует выбрать команду Open Project (Открыть проект). В результате на экран будет выдан блок диалога Load Project File (Загрузить файл проекта), который содержит перечень всех файлов с расширением имени *.PRJ в текущей директории, а также информацию о дате и времени создания первого файла в этом списке и его размере. Поскольку вы хотите создать новый файл, введите в блок ввода Load Project File (Загрузить файл проекта) имя MYPROG.

Заметим, что после открытия проекта в меню Project (Проект) становятся доступными команды Add Item (Добавить элемент), Delete Item (Удалить элемент), Local (Локальный) и Include Files (Включаемые файлы).

Замечание. Если файл проекта, который вы загружаете в память, находится в другой директории, то текущей директорией становится та, из которой был загружен файл проекта.

Вы можете держать свой файл проекта в любой директории. При загрузке такого файла проекта необходимо лишь указать в качестве составной части имени файла маршрутное имя. Если исходные файлы находятся в других директориях, вы должны также задать их маршрутные имена. Все имена файлов и соответствующие маршрутные имена задаются относительно той директории, из которой был загружен файл проекта.

После того, как вы ввели имя файла проекта, вы увидите окно Project (Проект).

Это окно содержит имя текущего файла проекта (MYPROG) и информацию относительно тех файлов, которые были Вами выбраны в качестве составных частей Вашего проекта. Для каждого файла будут отображаться его имя и маршрут. После компиляции этого файла будет также отображено число строк в файле и объем кода и данных в байтах, сгенерированных компилятором.

Строка состояния, отображаемая у нижнего края экрана, показывает, какое действие может быть выполнено в настоящий момент: по нажатию клавиши <F1> вы перейдете к системе подсказки, клавиша <Ins> - добавление файлов к проекту, <Del> - удаление файлов из проекта, <Ctrl+O> - задает параметры обработки файла <Пробел> - просмотр информации о включаемых файлах, относящихся к какому-либо файлу проекта, <F10> - перевод к основному меню. В данный момент нажмем клавишу <Ins>, чтобы добавить файл к списку файлов проекта.

Появляется блок диалога Add Item To Project List (Добавить элемент к списку файлов проекта); это блок диалога позволяет вам выбрать исходные файлы и добавить их к вашему проекту. Блок списка Files (Файлы) отображает все имена файлов в текущей директории, имеющие расширение имени *.C. В списке появятся файлы MYMAIN.C и MYFUNC.C. Доступны будут три кнопки действия: Add (Добавить), Cancel (Отменить) и Help (Подсказка).

*Замечание. Вы можете изменить спецификацию имени файла на необходимую вам с помощью блока ввода Name (Имя): по умолчанию используется маска *.C.*

Поскольку стандартной кнопкой является кнопка Add (Добавить), вы можете занести файл в окно Project (Проект), введя его имя в блок ввода Name (Имя) и нажав клавишу <Enter> или выбрав его имя в блоке списка Files (Файлы). Вы можете также осуществить поиск имени файла в блоке списка Files (Файлы), если введете несколько первых литер имени необходимого вам файла. В данном случае ввод символов должен привести к желаемому результату: маркер установится на имени MYFUNCS.C; нажмите клавишу <Enter>. Вы увидите, что имя MYFUNCS добавилось к окну Project (Проект), а вы вернулись к блоку диалога Add Item (Добавить элемент), где вы можете добавить другое имя файла. Таким же образом можно добавлять другие файлы. Система Borland C будет осуществлять компиляцию файлов именно в том порядке, в каком они появляются в проекте.

После того, как были заданы все параметры работы компилятора и имена директорий, система Borland C будет обладать всей информацией, которая ей необходима для построения программы MYPROG.EXE на основе кода, входящего в модули MYMAIN.C, MYFUNC.C и MYFUNC.H. Теперь перейдем к фактическому построению проекта.

Нажмите клавишу <F10>, чтобы перейти к основному меню. Теперь создайте программу MYPROG.EXE путем нажатия клавиши <F9> (или выбора команды Compile | Make EXE (Компилировать | Создать выполняемый файл)). Затем выполним программу путем нажатия клавиш <Ctrl+F9>. Для того, чтобы посмотреть выведенные вашей программой результаты, нажмите клавиши <Alt+F5>. Для возврата в интегрированную среду необходимо нажать любую клавишу на клавиатуре.

Когда вы покидаете интегрированную среду, проект, над которым вы работали, автоматически будет сохранен на диске. Сохраненный на диске проект будет состоять из двух файлов: файла проекта - с расширением *.PRJ и файла макета экрана - с расширением *.DSK. Файл проекта содержит информацию, которая необходима для построения ориентированного на проект выполняемого файла (с расширением имени *.EXE). Необходимая для построения выполняемого файла информация состоит из параметров работы компилятора, маршрутных имен включаемых файлов, файлов библиотек и результирующих файлов, параметров работы компоновщика, параметров выполнения избирательной компиляции и программ переноса. Файл макета экрана состоит из статусной информации по всем окнам на момент последнего использования данного проекта.

2.8. Система меню Borland C

2.8.1. Меню File(Файл)

Меню File (Файл) позволяет вам открывать и создавать файлы программ в окнах редактирования. Данное меню позволяет также сохранять внесенные изменения, выполнять другие действия над файлами, выходить в оболочку DOS и покидать систему Borland C.

Open (Открыть)

Команда отображает блок диалога, предназначенный для выбора файлов, чтобы вы могли выбрать файл программы, который будет открыт в окне редактирования.

Этот блок диалога содержит в себе блок ввода, блок списка файлов, кнопки с отметками Open (Открыть), Replace (Заменить), Cancel (Отменить), Help(Подсказка), а также информационную панель, которая описывает выбранный в настоящий момент файл.

По умолчанию действует кнопка Open, т.е. если вы введете имя полностью и нажмете клавишу <Enter>, система Borland C откроет указанный файл. Если вы вводите имя файла, который система Borland C обнаружить не может, она автоматически создаст и откроет новый файл с таким именем.

Блок списка отображает все имена файлов в текущей директории, которые соответствуют спецификациям в блоке ввода, отображает имя родительской директории, а также имена всех поддиректорий. Выбор файла осуществляется нажатием на клавиши ← ↑ → ↓. Фиксация выбора - нажатием клавиши <Enter>.

Для поиска имени файла можно ввести букву в нижнем регистре, а для поиска имени директории - букву в верхнем регистре.

Панель информации о файле, расположенная у нижнего края блока диалога Load a File (Загрузить файл), отображает маршрутное имя, имя файла, дату и время создания, а также размер выбранного вами в блоке списка файла.

New (Новый)

Команда позволяет открывать новое окно редактирования со стандартным именем NONAMExx.C (xx - число в диапазоне от 00 до 99). Эти файлы с родовым именем NONAME используются в качестве временного буфера для редактирования; когда вы сохраняете на диске файл с подобным именем, система Borland C запрашивает у вас действительное имя для этого файла.

Save (Сохранить)

Команда осуществляет запись на диск того файла, который находится в активном окне редактирования. Если файл сохраняется с именем NONAMExx.C, то система предоставляет возможность переименовать его и записать по новому пути (если это необходимо).

Save As (Сохранить под именем)

Команда осуществляет запись на диск того файла, который находится в активном окне редактирования под другим именем, в другой директории и на другом дисководе.

Save All (Сохранить все)

Команда действует аналогично команде Save с тем исключением, что она осуществляет запись на диск содержимого всех модифицированных файлов, а не только файла, находящегося в активном окне. Если ни одного окна редактирования не открыто, то эта команда переводится в режим «запрещена».

Print (Печать)

Команда позволяет вывести на печать содержимое активного окна редактирования. Данная команда будет «запрещена», если содержимое активного окна не может быть выведено на печать.

DOS Shell (Выход в оболочку DOS)

Команда позволяет временно выйти из системы Borland C, чтобы выполнить команду DOS или запустить какую-либо программу. Для того, чтобы вернуться в систему Borland C необходимо ввести команду EXIT и нажать на клавишу <Enter>.

Quit (Выйти)

Команда обеспечивает выход из системы Borland C, удаляет ее из памяти и возвращает вас к запросу со стороны DOS. Модифицированные и не сохраненные на диске файлы по запросу автоматически сохраняются.

2.8.2. Меню Edit (Редактирование)

Меню позволяет вам выполнять удаление, копирование и вставку текста в окна редактирования. Можно также открыть окно текстового буфера для просмотра или редактирования его содержимого. Команды этого блока меню становятся доступными только после выделения блока текста в окне редактора и можно использовать текстовый буфер. Данный буфер предназначен для удаления и вставки фрагментов текста. Он представляет собой специальное окно системы Borland C, хранящее тот фрагмент текста, который вы отсекли или скопировали, и который вы можете теперь куда угодно вставить или переместить.

Restore Line (Восстановить строку)

Команда отменяет действие последней команды редактирования, которая была применена к какой-либо строке. Данная команда действует только над последней модифицированной или отредактированной строкой.

Cut (Отсечь)

Команда удаляет выделенный фрагмент текста из документа и заносит его в текстовый буфер. Затем вы можете вставить текст в любой другой документ или в другое место текущего документа путем выбора команды Paste (Вклеить). Текст в текстовом буфере остается выделенным, поэтому вы можете вставлять его многократно.

Copy (Копировать)

Команда оставляет выделенный текст нетронутым, но заносит в текстовый буфер точную копию этого текста. Затем вы можете вставлять текст из буфера в любой другой документ. Можно скопировать текст из окна Help.

Paste (Вклеить)

Команда вставляет текст, расположенный в текстовом буфере, в текущее окно в ту позицию, где располагается курсор. Тот текст, который вы вставляете, представляет собой помеченный в настоящий момент фрагмент текста в окне ClipBoard.

Show ClipBoard (Показать содержимое текстового буфера)

Открывает окно текстового буфера, в котором хранятся фрагменты текста отсеченного и скопированного вами из других окон. Тот текст, который в настоящее время выделен, представляет собой тот текст, который система Borland C будет использовать при выборе команды Paste. Текст, размещенный в этом окне можно редактировать так, чтобы он полностью соответствовал вашим требованиям.

Clear (Стереть)

Команда удаляет выбранный фрагмент текста, но не заносит его в текстовый буфер. Это означает, что в дальнейшем вы не можете работать с этим куском текста.

3.Процесс проектирования

Рассмотрим вопросы посвященные разработке программ. Правила здесь довольно общего характера по своей природе, они совсем не затрагивают техники программирования на С или С++, а скорее рассматривают более общий процесс проектирования и разработки программы.

Эти правила относятся к процессу общего проектирования. Многие из них будут казаться банальными. Несмотря на это, некоторые из приводимых здесь правил являются самыми важными, потому что нарушение их может вызвать много бед, когда начнется собственно процесс разработки. В известном смысле, многие правила в этой главе предназначены для управленцев; программисты, хотя и знают о их существовании, редко имеют возможность применить свои знания на деле.

3.1. Сущность программирования: без сюрпризов, минимум сцепления и максимум согласованности

Многие (если не все) правила в этой книге могут (при желании) быть объединены в три метаправила, выраженные в заголовке этого раздела. Правило «без сюрпризов» само по себе не требует пояснений. Пользовательский интерфейс должен действовать так, как следует из его облика. Функция или переменная должны делать то, что означают их имена.

Сцепление — это связь между двумя программами или объектами пользовательского интерфейса. Когда один объект меняется, то все, с чем он соединен, может также измениться. Сцепление способствует появлению сюрпризов. (Я меняю эту штучку здесь, и внезапно вот та штуковина, в другом месте, перестает работать). Пример из С++: если объект одного класса посылает сообщение объекту второго класса, то посылающий класс сцеплен с принимающим классом. Если вы меняете интерфейс для принимающего класса, то вы также должны исследовать код в посылающем классе, чтобы убедиться в том, что он еще работает. Этот тип слабого сцепления безвреден. Для сопровождения программы вы должны знать об соотношениях в сцеплениях, но без некоторого количества сцеплений программа не могла бы работать. Несмотря на это, желательно, по мере возможности, минимизировать число соотношений сцепления.

Эта минимизация обычно выполняется в С посредством модулей, а в С++ — посредством классов. Функции в модуле(функции-члены в классе) сцеплены друг с другом, но за исключением нескольких интерфейсных функций (или объектов) они вовсе не сообщаются с внешним миром. В С вы должны использовать статический класс памяти, чтобы ограничить использование функции одним модулем. В С++ вы используете закрытые функции-члены.

Согласованность является противоположностью сцепления; группируемые вместе объекты (пункты диалогового и простого меню, функции в модуле, или члены класса), должны быть функционально связаны. Отсутствие связности также является «сюрпризом». В меню моего текстового редактора есть пункт «Настройка» и, кроме того, дополнительные опции настройки рассыпаны по четырем другим всплывающим меню. Я ожидал согласованной конфигурации и, когда не смог найти нужную мне опцию в пункте «Настройка», то решил, что этой опции просто нет. Эта плохо спроектированная система до сих пор раздражает меня; после года работы с ней я по-прежнему не помню, где расположена каждая опция, и часто со злостью вынужден тратить пять минут на поиск в пяти разных местах того, что хотел изменить. По отношению к исходному коду отсутствие согласованности заставляет вас делать то же самое — тратить свою жизнь на поиск объявлений функций в 15 различных файлах, что является очевидной проблемой при сопровождении.

3.2. Подавляйте демонов сложности

Ричард Рашид (разработчик Mach — варианта ОС UNIX)выступил несколько лет назад с фундаментальным обращением на конференции разработчиков Microsoft. Его главный смысл состоял в том, что слишком большая сложность, как в пользовательском интерфейсе, так и в программе является единственной крупной задачей, стоящей перед проектировщиками и пользователями программного обеспечения. По иронии судьбы, его речь была произнесена спустя два дня после провалившейся попытки показать нескольким тысячам очень толковых программистов, как следует программировать разработанный Microsoft интерфейс OLE 2.0 — один из самых сложных интерфейсов прикладного программирования, из когда-либо мною виденных. (OLE означает «связь и внедрение объектов». Стандарт OLE 2.0 определяет интерфейс, который может использоваться двумя программами для взаимодействия между собой определенным образом. Это действительно объектная ориентация на уровне операционной системы).

Предыдущий оратор, который убеждал пользоваться библиотекой Microsoft Foundation Class (MFC), сказал, что поддержка OLE в MFC «включает 20000 строк кода, необходимых для каждого базового приложения OLE 2.0». Аудитория была ошеломлена не полезностью MFC, а тем обстоятельством, что для написания базового приложения OLE 2.0 требуется 20000 строк кода. Любой такой сложный интерфейс

является ущербным. Следующие несколько правил используют OLE для иллюстрации характерных трудностей, но не подумайте, что проблема запутанности характерна лишь для Microsoft — она существует везде.

3.2.1. Не решайте проблем, которых не сушествует

3.2.2. Решайте конкретную проблему, а не общий случай

Поучительно воспользоваться OLE 2.0 в качестве примера того, что случается с многими, слишком сложными проектами. Сложность интерфейса OLE связана с двумя главными причинами. Во-первых, он безуспешно пытается быть независимым от языка программирования. Идея таблицы виртуальных функций C++ является центральной для OLE 2.0. Спецификация OLE даже пользуется нотацией классов C++ для документирования работы различных интерфейсов OLE. Для реализации OLE на другом языке программирования (не C++) вы должны имитировать на этом языке таблицу виртуальных функций C++, что фактически ограничивает ваш выбор языками C++, C или ассемблером (если вы не разработчик компиляторов, который может добавить к выбранному вами языку нужные свойства). Честно говоря, нужно быть сумасшедшим, чтобы программировать OLE не на C++; потребуется гораздо меньше времени на изучение C++, чем на написание имитатора C++. То есть, эта идея независимости от языка программирования является неудачной. Отказавшись отэтой идеи, интерфейс можно было бы существенно упростить.

Возвращаясь к истории из предыдущего раздела, нужно заметить, что библиотека MFC в действительности решает проблему сложности, связанную с OLE, при помощи простого и легко понятного интерфейса, реализующего все возможности, нужные для большинства приложений OLE 2.0. Тот факт, что никто не хотел программировать с OLE, пока для этого не появилась оболочка на основе MFC, впечатляет. Разработка хорошей оболочки вокруг плохого интерфейса не может быть решением лежащей в основе проблемы.

Если оболочка с использованием MFC столь проста, то почему же лежащий в ее основе пласт так сложен? Ответ на этот вопрос является основным предметом проектирования. Создатели интерфейса OLE никогда не задавали себе два основных вопроса:

- Какие основные функции должно поддерживать настоящее приложение?
- Как реализовать эти возможности простейшим способом?

Другими словами, они имели в виду не реальное приложение, когда они проектировали этот интерфейс, а какой-то худший в теоретическом смысле случай. Они реализовали самый общий интерфейс из возможных, не думая о том, что на самом деле предполагается делать при помощи этого интерфейса, и получили в результате систему, которая может делать все, но при этом оказалась слишком сложной для практического применения. (Вероятно, разработчики даже и не пробовали реализовать этот интерфейс в каком-либо приложении, иначе они бы обнаружили эти проблемы).

Процесс объектно-ориентированного проектирования является в какой-то мере попыткой решения этой проблемы. Относительно просто добавить новую возможность в объектно-ориентированную систему или посредством наследования, или добавив новые обработчики сообщений к существующим классам. Скрывая определения данных от пользователя класса, вы оставляете за собой право полностью менять внутреннюю организацию класса, включая определения данных, не беспокоя пользователей этого класса, при условии, что вы сохраняете существующий интерфейс.

В структурном проектировании вы не позволите себе подобную роскошь. Вначале вы обычно проектируете структуры данных, и ваша главная задача — это модификация структуры данных, потому что нужно проверить каждую подпрограмму, использующую эту структуру, чтобы убедиться в том, что она еще работает. В результате «структурные» программы зачастую содержат массу бесполезного кода. Это делается в надежде на то, что кто-либо в будущем захочет воспользоваться той или иной функцией. На деле многие проектанты структурных программ гордятся своей способностью предсказывать направление, в котором может развиваться программа. В целом, все это вызывает массу бесполезного труда и приводит к программам с большим, чем требуется, размером.

Вместо того, чтобы учитывать в проекте все возможные случаи, пишите свой код таким образом, чтобы его легко можно было расширить в случае реальной необходимости•добавить новые функции. В этих случаях объектноориентированные проекты, как правило, работают лучше.

3.3. Интерфейс пользователя не должен напоминать компьютерную программу (принцип прозрачности)

Однажды я услышал от кого-то, что лучшим из когда-либо разработанных пользовательских интерфейсов является карандаш. Его назначение тотчас же понятно, он не нуждается в руководстве пользователя, он позволяет решить задачу без особого напряжения. Однако, наиболее важным его

свойством является прозрачность. Когда вы пользуетесь карандашом, то думаете лишь о том, что пишете, а не о самом карандаше.

Подобно карандашу, лучшими компьютерными интерфейсами являются те из них, которые скрывают сам факт обращения к компьютеру: в этом смысле интерфейс с системой зажигания вашего автомобиля представляет собой выдающийся пример. Поворачиваете зажигание, включаете скорость и жмете на газ, как если бы все эти объекты интерфейса (ключ, рычаг скоростей, педаль) были прицеплены прямо на двигатель. Тем не менее, это не так: они всего-навсего простые устройства ввода в компьютер, который управляет двигателем.

К сожалению, подобный уровень четкости восприятия зачастую отсутствует в пользовательских интерфейсах. Представьте себе графический интерфейс пользователя Windows на автомобиле. Вы трогаетесь, выбрав в главном меню пункт «Движение автомобиля». Щелчком должно открыться меню «Переключение скорости», которое предложит набор опций «Вперед», «Назад» и «Нейтраль». Щелкните мышью на одной из них для выбора нужного направления. Затем вернитесь в меню «Движение автомобиля» и выберите команду «Поехали». Появится диалоговое окно «Скорость», где вы должны воспользоваться ползунком для ввода желаемой скорости. Однако, установить правильную скорость не так легко из-за грубого разрешения ползунка (пол-миллиметра на мышке дает около 1 км/ч), поэтому, скорее всего, вы установите 59,7 км/ч, вместо 60. Затем в диалоговом окне вы нажимаете кнопку «Поехали», вслед за чем появляется сообщение «Ручной тормоз стоянки не убран — нажмите Fl для справки» (динамик издает громкий звук). Покорно щелкаете кнопкой «ОК», чтобы убрать окно сообщений, затем снова пытаетесь открыть главное меню, но машина просто посылает вам звуковой сигнал. Наконец, поняв, что дело в том, что диалоговое окно «Скорость» еще отображается, вы щелкаете на кнопке «Отмена», чтобы убрать его. Вы открываете меню: «Ручной тормоз на стоянке» и убираете флажок «Включен».

Затем вы снова открываете окно «Поехали». И вновь получаете сообщение (и громкий звук) о том, что вы должны сначала выбрать направление в меню «Переключение скорости». В этот момент вы решаете, что лучше стоило бы пройтись на работу пешком.

Вот вам другой пример: занимаясь недавно подготовкой обзора, я просмотрел несколько программ авиационных бортовых журналов. («Бортовой журнал» — это очень простой табличный документ. Каждая строка соответствует отдельному полету, а столбцы разбивают общую продолжительность полета по категориям: итоговая продолжительность, продолжительность полета в облаках и т.п. В других столбцах полет помечается как деловой и так далее).

До сих пор самым лучший из интерфейсов всегда выгляел совершенно одинаково, как привычный бумажный журнал, но он автоматизировал нудную работу. Вы вводили время в «итоговый» столбец — и то же самое время появлялось в других подходящих по смыслу столбцах. Значения по столбцам складывались автоматически для получения итогов по категориям. Вы могли легко генерировать необходимые отчеты и экспортировать данные в формат ASCII с разделителями из символов табуляции, который читается поистине любыми в мире программами электронных таблиц или подготовки текстов. Для непривычного взгляда весь интерфейс казался, мягко говоря, разочаровывающим, но он был функциональным и интуитивно понятным, а программа — короткой и быстрой. Однако самым важным было то, что этот интерфейс выглядел как бортовой журнал, а не как программа для Windows.

Другой крайностью был ошеломляющий графический интерфейс пользователя (GUI) под Windows: у него были диалоговые окна; у него была трехмерная графика; вы могли генерировать круговые диаграммы, показывающие процент продолжительности полета в облаках по отношению к вашему общему налету на «Цесне-172» за последние 17 лет; вы могли помещать внутри отсканированную фотографию самолета...- вот вам картина! Программа выглядела превосходно, но использовать ее было почти невозможно. Практической необходимости для создания большинства из генерируемых ей диаграмм и отчетов не было. Ввод данных был неудобным и медленным — нужно было вызвать диалоговое окно с полями, разбросанными по всей его поверхности. Фактически вы должны были прочитать все, чтобы обнаружить ту категорию, которая вас интересовала, а некоторые из категорий были скрыты за кнопками, с неизбежностью приводя к сложному поиску. Чтобы добавить еще каплю дегтя, скажу, что эта программа была надстроена над сервером реляционной базы данных (помните, что это для поддержки простой таблицы без реляционных связей). Она заняла 30 Мбайт на моем диске. Почти пять минут уходило у меня на запись, которая требовала около 10 секунд на листах бортового журнала или упомянутом ранее простом графическом интерфейсе пользователя. Программа была бесполезна, но, конечно, потрясающа.

Одна из главных трудностей состояла в том, что инструментарий для второй программы определяет проектирование интерфейса. Все эти программы были разработаны на Visual Basic, языке очень высокого уровня (который мне, между прочим, по сути дела, не сильно нравится). Приложения, созданные при помощи таких генераторов приложений, как Visual Basic (или Power Builder, или Delphi, или ...), обычно имеют специфический внешний вид, который незамедлительно говорит об инструменте, использованном для построения этого приложения. Разработчик интерфейса не может надеяться на помощь, если этот специфический облик не подходит для конкретного проекта. Пользователи генераторов приложений

должны на выбор иметь несколько вариантов, чтобы затем использовать тот генератор, который лучше всего соответствует требованиям данного интерфейса. Несмотря на это, опыт мой показывает, что наиболее приемлемые программы со временем должны перенести по меньшей мере часть кода из интерфейса на язык низкого уровня типа C или C++; поэтому важно, чтобы ваш генератор приложений позволял использовать и код низкого уровня.

3.4. Не путайте легкость в изучении с легкостью в использовании

Эта проблема когда-то касалась почти исключительно машин Macintosh, но Windows и здесь в последнее время выходит вперед. Компьютер Mac был спроектирован так, чтобы прежде всего оказаться простым в освоении. Положим, что тетушка Матильда Мак-Гиликатти часто заходила в компьютерный магазин, чтобы пользоваться предлагаемыми услугами мгновенной распечатки кулинарных рецептов. В итоге Матильда забирает компьютер домой и успешно вводит реценты в течение нескольких месяцев. Теперь она хочет взять эти рецепты, проанализировать их химический состав и написать статью в научный журнал о коллоидных свойствах продуктов питания на основе альбумина. Доктор Мак-Гиликатти — хорошая машинистка, печатающая обычно около 100 слов в минуту, но эта ужасная мышь ее постоянно тормозит. Каждый раз, когда ее руки отрываются от клавиатуры, она теряет несколько секунд. Она пытается найти слово в своем документе и обнаруживает, что для этого она должна открыть меню, ввести текст в диалоговое окно и щелкнуть по нескольким экранным кнопкам. В конце файла она должна явно указать утилите поиска возвратиться к его началу. (Ее версия редактора vi 15-летней давности позволяет выполнить все это при помощи двух нажатий клавиш — без необходимости отрываться от клавиатуры). Наконец, она обнаруживает, что на выполнение обычной работы — подготовки статьи в журнал — уходит в два раза больше времени, чем раньше, в основном из-за проблем интерфейса пользователя. Ей не понадобилось руководство, чтобы пользоваться этой программой, — ну и что?

Вернемся к примеру с карандашом из предыдущего параграфа. Карандаш — это трудная задача. У большинства детей он отнимает несколько лет. (Можете возразить, что, судя по каракулям на рецептах, многие врачи так этому и не научились). С другой стороны, после приобретения навыков, вы обнаружили, что карандашом пользоваться очень легко.

Самое главное здесь в том, что опытному пользователю зачастую нужен совершенно другой, чем начинающему интерфейс. Вспомогательные средства типа «горячих» клавиш не разрешают эту проблему; старый неуклюжий интерфейс пользователя все-таки препятствует производительности и, на самом деле, нет разницы, откроете ли вы меню с помощью «горячей» клавиши или мышью. Все трудности самом меню.

3.5. Производительность может измеряться числом нажатий клавиш

Интерфейс, требующий меньше нажатий клавиш (или других действий пользователя типа щелчков мышью), лучше того, который требует много нажатий для выполнения одной и той же операции, даже если такие виды интерфейсов обычно сложнее в освоении.

Подобным образом пользовательскими интерфейсами скрывающими информацию в меню или за экранными кнопками, обыкновенно труднее пользоваться, потому что для выполнения одной задачи необходимо выполнить несколько операций (вызвав подряд несколько спускающихся меню)

Хороший пример — настройка программы. Во многих используемых мной ежедневно программ опции настройки рассыпаны по нескольким меню. То есть для вызова диалогового окна, настраивающего один из аспектов программы (например, выбор шрифта), я должен взять одно меню. Затем я должен вызвать другое меню, чтобы сделать что-то в том же духе (например, выбрать цвет). Лучше поместить все опции настройки на одном экране и использовать форматирование экрана для объединения опций по назначению.

3.6. Если вы не можете выразить что-то на повседневном языке, то вы не сможете сделать это и на C/C++

Это правило, наряду с последующим, также относится правилам пользовательского интерфейса, но здесь под «пользователем» уже понимается программист, использующий написанный вами код — зачастую это вы сами.

Акт записи на обычном языке описания того, что делает программа, и что делает каждая функция в программе, является критическим шагом в мыслительном процессе. Хорошо построенное, грамматически правильное предложение - признак ясного мышления. Если вы не можете это записать, то велика вероятность того, что вы не полностью продумали задачу или метод ее решения. Плохая грамматика и построение предложения являются также показателем поверхностного мышления. Поэтому, первый шаг в написании любой программы — записать, что именно и как делает программа.

Есть разные мнения о возможности мышления вне языка, но я убежден, что аналитическое мышление того типа, который нужен в компьютерном программировании, тесно связано с языковыми навыками. Я не думаю, что является случайностью то, что многие из знакомых мне лучших программистов имеют

дипломы по истории, филологии и схожим наукам. Также не является совпадением то, что некоторые из виденных мной худших программ были написаны инженерами, физиками и математиками, затратившими в университете массу энергии на то, чтобы держаться как можно дальше от занятий по языку и литературе.

Сущность заключается в том, что математическая подготовка почти не нужна в компьютерном программировании.

Тот тип организационного мастерства и аналитических способностей, который нужен для программирования, связан полностью с гуманитарными науками. Логика, например, во время моей учебы преподавалась на философском факультете. Процесс, используемый при проектировании и написании компьютерных программ, почти полностью идентичен тому, который используется при сочинении музыки и написании книг. Процесс программирования вовсе не связан с теми процессами, которые используются для решения математических уравнений.

Здесь я делаю различие между информатикой (computer science) — математическим анализом компьютерных программ — и программированием или разработкой программного обеспечения — дисциплиной, интересующейся написанием компьютерных программ. Программирование требует организационных способностей и языковой подготовки, а не абстрактного мышления, необходимого для занятий математическим анализом. (В университете меня заставили целый год посещать лекции по математическому анализу, но я никогда из него ничего не использовал ни на курсах по информатике, хотя для этих курсов матанализ был необходимым условием, ни в реальной жизни).

Как-то я получил открытую рецензию на мою книгу, посвященную предмету проектирования компиляторов, в которой рецензент (преподаватель одного из ведущих университетов) заявил, что «считает абсолютно неуместным включение исходного кода компилятора в книгу о проектировании компиляторов». По его мнению, необходимо учится «фундаментальным принципам» — лежащей в основе математики и теории языка, а детали реализации «тривиальны». Первое замечание имеет смысл, если у вас создалось впечатление, что книга написана ученым-специалистом по информатике, а не программистом. Рецензент интересовался лишь анализом компилятора, а не тем как его написать. Второе замечание просто показывает вами, насколько изолировала себя научная элита от реального труда программирования. Интересно, что основополагающая работа по теории языка, сделавшая возможным написания компиляторов, была выполнена в Массачусетском технологическом институте (MIT) лингвистом Наумом Хомским, а не математиком.

Обратной стороной медали является то, что, если вы зашли в тупик при решении проблемы, то лучший способ выйти из него — разъяснить задачу приятелю. Почти всегда решение возникает в вашей голове в ходе объяснения.

3.6.1. Начинайте с комментариев

Если вы последовали совету в предыдущем правиле, то комментарии для вашей программы уже готовы. Просто возьмите описание по использованию, которое вы только что написали и добавьте вслед за каждым абзацем блоки кода, реализующие функции, описанные в этом абзаце. Оправдание «у меня не было времени, чтобы добавить комментарии» на самом деле означает — «я писал этот код без проекта системы и у меня нет времени воспроизвести его». Если создатель программы не может воспроизвести проект, то кто же тогда сможет.

3.7. Читайте код

Все писатели — это читатели. Вы учитесь, когда смотрите, что делают другие писатели. Удивительно, но программисы — писатели на C++ и C — часто не читают код. Тем хуже. Я настоятельно рекомендую, чтобы, как минимум, члены группы программирования читали код друг у друга. Читатель может найти ошибки, которые вы не увидели, и подать мысль, как улучшить код.

Идея здесь — не формальная «критика кода», имеющая довольно сомнительный характер: никто не хочет наступать на ногу коллеге, поэтому шансы получить полезную обратную связь в формальной ситуации малы. Для вас лучше присесть с коллегой и просто разобрать код строка за строкой, объясняя что как делается и получая какую-то обратную связь и совет. Для того, чтобы подобное упражнение принесло пользу, автор кода не должен делать никаких предварительных пояснений. Читатель должен быть способен понимать код в процессе чтения. (Всем нам приходилось иметь дело с учебниками, столь трудными для понимания, что нельзя было ни в чем разобраться без объяснения преподавателя. Хотя это и гарантирует, что преподаватель не останется без работы, но никак не отражается на авторе учебника.)

Если вам пришлось объяснять что-то вашему читателю, то это значит, что ваше объяснение должно быть в коде в качестве комментария. Добавьте этот комментарий, как только вы его произнесли; не откладывайте этого до окончания просмотра.

3.7.1. В цехе современных программистов нет места примадоннам

Это следствие правила чтения. Программисты, которые думают, что их код совершенен, которые отвергают критику, вместо того, чтобы считать ее полезной, и которые настаивают на том, что они должны работать 'втихомолку, вероятно, пишут тарабарщину, не поддающуюся сопровождению даже если кажется, что она работает.

3.8. Разлагайте сложные проблемы на задачи меньшего размера

На самом деле это также и правило литературного стиля. Если очень трудно объяснить точку зрения за один раз, то разбейте изложение на меньшие части и по очереди объясняйте каждую. То же самое назначение у глав в книге и параграфов в главе.

В качестве примера программирования возьмем прошитое бинарное дерево, отличающееся от нормального дерева тем что указатели на узлы-потомки в конечных узлах на листочках указывают на само дерево. Фактическим преимуществом прошитого дерева является то, что его легко просмотреть нерекурсивно при помощи этих дополнительных указателей. Проблема заключается в том, что сложно выйти из алгоритмов просмотра (в особенности при обратном просмотре). С другой стороны, имея указатель на узел, легко написать алгоритм поиска последующего элемента в обратном порядке. Путем изменения формулировки с «выполнить просмотр в обратном порядке» на «начав с самого отдаленного узла, искать последующие элементы в обратном порядке, пока они не закончатся» получаем разрешимую задачу.

3.9. Используйте язык полностью

3.9.1. Используйте для работы соответствуюший инструмент

Данное правило является спутником правила «Не путайте привычность с читаемостью», представленного ниже, но, скорее всего, больше касается проблем руководства. Мне часто говорят, что студентам не разрешается использовать некоторые аспекты С или С++ (обычно это указатели), потому что они «нечитабельны». Обычно это правило навязывается руководителями, знающими ФОРТРАН, БЕЙСИК или какой-то другой язык, не поддерживающий указатели, и их не очень-то заставишь изучать С. Вместо того, чтобы признать изъяны в своих знаниях, такие руководители будут предпочитать калечить своих программистов. Указатели превосходно читаются программистами на С.

И, наоборот, я видел ситуации, где руководство требовало, чтобы программисты перешли с языка программирования типа КОБОЛ на С, но не желало оплачивать переподготовку, необходимую для перехода. Или еще хуже, руководство платило за переподготовку, но не предоставляло времени, необходимого для действительного изучения материала. Переподготовка является занятием, требующим всего рабочего дня. Вы не можете одновременно выполнять «полезную» работу, а если попытаетесь, то ваши деньги будут выброшены на ветер. Так или иначе, после того, как руководители видят, что их штат не был превращен в гуру программирования на С++ после 3-дневного краткого курса, они реагируют наложением ограничений на использование некоторых компонентов языка. Фактически они говорят: «Вы не можете использовать ту часть С++, которая не похожа на язык, который мы использовали до перехода на С++». Естественно, что окажется невозможным эксплуатировать ни одну из прогрессивных особенностей языка (которые прежде всего и являются главной причиной его использования), если вы ограничите себя «простейшим» подмножеством особенностей.

Глядя на эти ограничения, мне в первую очередь интересно знать, зачем понадобилось менять КОБОЛ на С. Принуждение программистов на языке КОБОЛ использовать С всегда поражало меня своей большой глупостью. КОБОЛ — великолепный язык для работы с базами данных. У него есть встроенные примитивы, упрощающие выполнение задач, которые довольно трудны для С. С, в конце концов, был разработан для создания операционных систем, а не приложений баз данных. Довольно просто дополнить КОБОЛ, чтобы он поддерживал модный графический интерфейс пользователя, если это единственная причина перехода на С.

3.10. Проблема должна быть хорошо продумана перед тем, как она сможет быть решена

Эти правила посвящены весьма реальным проблемам и во многих отношениях являются самыми важными правилами в этой книге.

Настоящее правило является настолько очевидным утверждением в повседневной жизни, что кажется странным его восприятие едва ли не ересью в применении к программированию. Мне часто говорят, что «невозможно потратить пять месяцев на проектирование, не написав ни одной строки, кода — ведь наша

производительность измеряется числом строк кода, написанных за день». Люди, говорящее это, обычно знают, как делается хороший проект; просто они не, могут позволить себе такую «роскошь».

Мой опыт говорит, что хорошо спроектированная программа не только работает лучше (или просто работает), но и может быть написана быстрее и быть проще в сопровождении, чем плохо спроектированная. Лишние четыре месяца при проектировании могут сэкономить вам более четырех месяцев на этапе реализации и буквально годы в период сопровождения. Вы не сможете добиться высокой производительности, если приходится выбрасывать прошлогоднюю работу из-за существенных изъянов проекта.

Кроме того, скверно спроектированные программы труднее реализовать. Тот аргумент, что у вас нет времени на проектирование, потому что вы «должны захватить рынок программ как можно скорее», просто не выдерживает никакой критики, потому что реализация плохого (или никакого) проекта требует гораздо больше времени.

3.11. Компьютерное программирование является индустрией обслуживания

Меня иногда шокирует неуважение, проявляемое некоторыми программистами по отношению к пользователям своих программ, как если бы «пользователь» (произносится с презрительной усмешкой) был низшей формой жизни, неспособной к познавательной деятельности. Но факт состоит в том, что весь компьютер существует лишь с одной целью: служить конечному пользователю наших продуктов. Если никто бы не пользовался компьютерными программами, то не было бы и программистов.

Печальным фактом является то, что существенно больше половины из ежегодно разрабатываемого кода выбрасывается за ненадобностью. Такие программы или никогда не поступают в эксплуатацию, или используются лишь очень короткое время, после чего выбрасываются. Это означает невероятную потерю производительности, сокращая для большинства управленцев реальные среднесуточные цифры выработки. Подумайте о всех начинающих фирмах, выпускающих программы, которые никогда не будут проданы, о всех группах разработчиков, пишущих бухгалтерские пакеты, которыми нельзя воспользоваться.

Легко увидеть, как возникает эта печальная ситуация: программисты создают программы, которые никому не нужны. Исправить ее тоже легко, хотя в определенном окружении это и сталкивается с неожиданными трудностями: спросите людей, что им нужно, и затем сделайте то, что они вам сказали.

К сожалению, многие программисты производят впечатление лиц, полагающих, что конечные пользователи не знают чего хотят. Вздор. Почти всегда пользователи оказываются так запуганы сыплющим специальными терминами «экспертом», что замолкают. Мне часто говорили: «Я знаю, что мне нужно, но не могу это выразить». Лучший ответ на это: «Отлично, скажите это на нормальном языке — я сделаю перевод на компьютерный».

3.12. Вовлекайте пользователей в процесс проектирования

3.13. Заказчик всегда прав

Ни одной программе не добиться успеха, если ее проектировщики не общаются непосредственно с ее конечными пользователями. Несмотря на это, часто ситуация больше напоминает игру («испорченный телефон»), в которую многие из нас играли в детском саду, когда 20 ребятишек садятся в кружок. Кто-нибудь шепчет фразу своему соседу (соседке), который передает ее своему, и так далее по кругу. Забава заключается в том, чтобы послушать как сообщение звучит после того, как пройдет весь круг — обычно ничего похожего на исходную фразу. Тот же самый процесс зачастую встречается при разработке программ. Пользователь говорит с управляющим, докладывающим другому управляющему, который нанимает консультационную фирму. Президент консультационной фирмы разговаривает с руководителем разработчиков, который в свою очередь говорит со старшим группы, обращающимся наконец к программистам. Шансы на то, что даже простой документ с требованиями останется после этого процесса невредимым, равны нулю. Единственным решением этой проблемы является тесное вовлечение пользователей в процесс разработки, лучше всего путем включения по крайней мере одного конечного пользователя в команду разработчиков.

Родственная ситуация складывается в случае простой самонадеянности части программистов, которые говорят: «Я знаю, что пользователи сказали, что им нужно сделать это таким-то способом, но у них нет достаточных знаний о компьютерах, чтобы принять сознательное решение; мой способ лучше». Такое отношение фактически гарантирует, что программой никогда не будут пользоваться. Исправить ситуацию здесь можно, официально назначив конечного пользователя лицом, оценивающим качество проекта. Никто не может начать писать код до тех пор, пока пользователь-член команды не даст на это добро. Сотрудники, игнорирующие проект в пользу своих идей, должны быть уволены. В реальной жизни для подобного типа детского упрямства на самом деле нет места.

При этом нужно сказать, что опытный проектировщик зачастую предлагает лучшее решение проблемы, чем придуманное конечным пользователем, в особенности, если учесть, что конечные пользователи часто предлагают интерфейсы, созданные по образцу программ, которыми они постоянно пользуются. Несмотря на это, до начала реализации вы должны убедить пользоваться в том, что ваш способ лучше. «Лучший» интерфейс не является лучшим, если никто, кроме вас, не сможет (или не захочет) им пользоваться.

3.14. Малое это прекрасно. (Большое == медленное)

Распухание программ является огромной проблемой. В стародавние времена ОС'ы работали на 16-разрядной шине с 64Кбайтами внутренней памяти. В наше время большинство операционных систем требуют 32-разрядных машин с минимум 16 Мбайтами оперативной памяти, чтобы работать с приемлемой скоростью. Очевидно, что большая часть этого распухания памяти является результатом небрежного программирования.

В добавок к проблеме размера у вас также есть и проблема со временем выполнения. Виртуальная память не является настоящей памятью. Если ваша программа слишком велика, чтобы поместиться в оперативной памяти, или, если она выполняется одновременно с другими программами, то она должна периодически подкачиваться с диска. На эти подкачки, мягко выражаясь, расходуется время. Чем меньше программа, тем менее вероятно, что произойдет подкачка, и тем быстрее она будет выполняться.

Третьей проблемой является модульность. Одно из фундаментальных положений гласит — «меньше — лучше». Большие задачи лучше выполняются взаимодействующей системой небольших модульных программ, каждая из которых хорошо исполняет лишь одно задание, но каждая из них может сообщаться с другими компонентами. (Стандарт связи и внедрения объектов Microsoft (OLE) добавляет это свойство в Windows, а OpenDoc — в Macintosh.) Если ваше приложение представляет собой модульную конструкцию из маленьких программ, работающих вместе, то вашу программу очень просто настраивать по заказу путем смены модулей. Если вам не нравится этот редактор, то поменяйте его на новый.

Наконец, программы обычно уменьшаются в процессе усовершенствования. Большие программы, вероятно, никогда не подвергались усовершенствованиям.

В поисках решения этой трудности обнаружено, что коллективы программистов с плохим руководством часто создают излишне большие программы. То есть, группа ковбоев от программирования, каждый из которых работает в одиночку в своем офисе и не разговаривает друг с другом, напишет массу лишнего кода. Вместо одной версии простой служебной функции, используемой по всей системе, каждый программист создаст свою версию для одной и той же функции.

3.15. Прежде всего, не навреди

Это правило касается сопровождения программ. Известно, что в больших компьютерных программах стоит тронуть что-то, кажущееся маловажным, и сразу же весь ее код перестает работать. Объектно-ориентированные методы разработки программ прежде всего служат для решения (или по крайней мере для облегчения решения) этой проблемы в будущем, но ведь есть уже миллионы строк старых кодов, которые сегодня требуют сопровождения.

Иногда программисты изменяют части программ лишь только потому, что им не нравится как выглядит ее код. Это плохо. Если вы не знаете всех частей программы, затрагиваемых изменением (а это почти невозможно), то не трогайте код. Вы можете вполне резонно возразить, что на самом деле ни одно из, изложенных выше, не относится к сопровождению программ. Вы просто не сможете изменить существующий код программы в соответствии с каким-то методическим руководством (как бы вам этого ни хотелось), без риска нанести ей непоправимый ущерб. Предлагаемые здесь правила полезны лишь только тогда, когда у вас есть блестящая возможность начать программу с нуля.

3.16. Отредактируйте свой код

3.17. Программа должна писаться не менее двух раз

3.18. Нельзя измерять свою производительность числом строк

Раньше, когда вы изучали в школе литературу, вам никогда не приходило в голову сдавать черновик письменного задания, если вы, конечно, рассчитывали на оценку выше тройки. Тем не менее, многие компьютерные программы являются просто черновиками и содержат столько же ошибок сколько и черновики ваших сочинений. Хороший код программы сначала написан, а затем отредактирован в целях улучшения. (Конечно, я имею в виду «редактировать» в смысле «исправлять».)

Учтите, что редактирование должно быть сделано своевременно, потому что неотредактированный текст программы по сути невозможно сопровождать (точно также, как и ваше неотредактированное сочинение было бы невозможно прочесть). Авторы программы знакомы с ее кодом и могут выполнить

редактирование более эффективно, чем программист, занимающийся сопровождением, который сначала должен ее расшифровать, прежде чем окажется возможным выполнить какую-либо реальную работу.

К сожалению, в отчетных документах это выглядит впечатляюще, когда кто-то пишет программу быстро, но не думает при этом о ее сопровождении или элегантности. «Ого, она выдает в два раза больше кода вдвое быстрее». Учтите, что какой-то несчастный программист, сопровождающий задачу, будет затем вынужден потратить в восемь раз больше времени, сокращая первоначальный размер программы наполовину и делая ее пригодной для использования. Число строк кода в день, как мера объема, не является мерилом производительности.

3.19. Вы не можете программировать в изоляции

В классической книге Джеральда Уэйнберга Психология программирования для компьютеров (The Psychology of Computer Programming, New York, Van Nostrand Reinhold, 1971) приводится прелестная история об автоматах с газированной водой. Администрация одного вычислительного центра решила, что сотрудники тратят массу времени у автоматов с газированной водой. Люди создают много шума и ничего при этом не делают, поэтому автоматы убрали. Через несколько дней консультанты на местах были настолько перегружены работой, что к ним стало невозможно обратиться. Мораль в том, что люди вовсе не зря тратили время; оказывается, издавая весь этот шум, они помогали друг другу в решении проблем. Изоляция может стать настоящей проблемой в группе объектно-ориентированного проектирования, которая обязательно должна состоять из пользователей, проектировщиков-программистов, специалистов по документации и т.д., работающих совместно. Так как число программистов в этой группе зачастую меньше, чем в более традиционных проектных коллективах, то становится трудно найти кого-то, с кем можно обсудить проблемы; производительность страдает. Подумайте о еженедельных дружеских вечеринках, как средстве повышения производительности.

3.20. Прочь глупости

Если вы не можете решить трудную задачу, займитесь на некоторое время чем-либо другим. Программисты зачастую наиболее производительны, когда смотрят в окно, слоняются по коридорам с пустым выражением лица, сидят в кафе и пьют кофе с молоком, или как-то еще «теряют время».

3.21. Пишите программу с учетом сопровождения — сопровождающим программистом являетесь вы сами

Сопровождение начинается немедленно после написания кода программы, а сопровождением на этой стадии обычно занимаетесь вы сами. Это хорошая мысль — осчастливить сопровождающего программиста. Поэтому ваша первая забота состоит в том, чтобы программа легко читалась. Структура и назначение каждой строки должны быть всеобъемлюще ясны, а если это не так, то следует добавить поясняющие комментарии.

Одной из причин того, что математические доказательства корректности программ остаются донкихотством, является то, что программ – без ошибок не бывает. Каждая программа не только содержит ошибки, но и требования к ней меняются сразу же с момента ее эксплуатации и у пользователя появляются потребность в каких-то новых свойствах, что вызывает появление новых и усовершенствованных ошибок. Так как ошибки всегда с нами, то мы должны писать наш программный код так, чтобы ошибки всегда можно было легко искать. Вы можете переформулировать это правило: Не умничайте. Изощренный код никогда нельзя сопровождать. Очевидно, что ваша программа непременно должна быть максимально эффективной, но первая из ваших задач — сопровождение, и вы не должны приносить читабельность на алтарь эффективности. Сначала напишите программу с учетом сопровождения, затем запустите отладчик для своей программы и определите ее узкие места. Вооруженные реальной информацией, вы уже знаете, где подменить читаемость скоростью, и можете вернуться и внести изменения.Сохраняйте первоначальный текст в комментариях, либо весь модуль до изменения, чтобы, в случае необходимости, можно было бы вернуться назад. Всегда помните, что любые манипуляции с текстом программы не повысят эффективность в той мере, как это сделает лучший алгоритм. Простой пример - пузырьковая сортировка идет медленно, вне зависимости от того, насколько хорошо написан код.

4. Язык программирования С

4.1. Символика языка Си

Как и любой другой алгоритмический язык Си базируется на трех китах, соответственно представляющих:

- изобразительные средства языка – алфавит;
- объекты – данные разного типа;
- процедуры обработки данных – операторы.

Все это, естественно, дополняется синтаксисом языка – сводом правил по написанию предложений (строк программы) и оформлению программных модулей.

Алфавит Си в большинстве своем совпадает с алфавитом других алгоритмических языков программирования, т.к. он продиктован спецификой клавиатуры ПК. В его состав входят большие и малые буквы латинского алфавита, цифры и различные разделители – скобки, знаки препинания и другие отображаемые символы клавиатуры. Буквы русского языка, как и в большинстве языков, имеют ограниченное применение – как правило, для описания значений текстовых констант и символьных переменных. Кроме того, в Си довольно своеобразно используются и трактуются некоторые символы и их сочетания. Более подробная информация об этом приведена в табл. 1 – 4.

Таблица 1

Операции над одним операндом	
Обозначение операции	Пояснение
&x	адрес переменной x
*p	имя переменной, на которую смотрит указатель p
–x	смена знака значения переменной x
+x	редко используемая операция, не меняющая значения x
~x	инвертирование битов целочисленного значения x
!x	отрицание значения x (0 или не 0)
+ +x	увеличение значения x на 1 до его использования в последующих вычислениях
x+ +	увеличение значения x на 1 после его использования в последующих вычислениях
– –x	уменьшение значения x на 1 до его использования в последующих вычислениях
x– –	уменьшение значения x на 1 после его использования в последующих вычислениях
sizeof x	определение размера значения переменной x в байтах

Таблица 2

Операции над двумя операндами	
Обозначение операции	Пояснение
a + b	сложение числовых данных
a – b	вычитание числовых данных
*a * b*	умножение числовых данных
a / b	деление числовых данных. Если операнды целочисленные, то дробная часть результата теряется.
a % b	остаток от деления целочисленных данных
a << k	сдвиг значения *a* на *k* двоичных разрядов влево (аналог умножения *a* на 2^k)
a >> k	сдвиг значения *a* на *k* двоичных разрядов вправо (аналог деления *a* на 2^k)
a & b	поразрядное логическое умножение целочисленных операндов (операция «И»)
a | b	поразрядное логическое сложение целочисленных операндов (операция «ИЛИ»)
a ^ b	операция «исключающее ИЛИ» над целочисленными операндами
a < b	сравнение числовых или символьных данных на «меньше»
a <= b	сравнение числовых или символьных данных на «меньше или равно»
a > b	сравнение числовых или символьных данных на «больше»
a >= b	сравнение числовых или символьных данных на «больше или равно»
a == b	сравнение числовых или символьных данных на «равно»
a != b	сравнение числовых или символьных данных на «не равно»
y1 && y2	проверка одновременного выполнения условий *y1* и *y2*
y1 || y2	проверка выполнения хотя бы одного из условий *y1* или *y2*
p + int	прибавление к адресу (указатель *p*) целочисленного смещения int*sizeof, определяемого

	типом данных, на которые смотрит указатель
p – int	вычитание из адреса (указатель *p*) целочисленного смещения int*sizeof, определяемого типом данных, на которые смотрит указатель

Таблица 3

Операции присваивания	
Обозначение операции	Пояснение
v = e	присвоить переменной *v* значение выражения *e*
v1=v2=...=e	множественное присвоение значения выражения e переменным v1,v2,...
a += b	аналог оператора *a = a + b*
a –= b	аналог оператора *a = a – b*
*a *= b*	аналог оператора *a = a * b*
a /= b	аналог оператора *a = a / b*
a %= b	аналог оператора *a = a % b*
a <<= k	аналог оператора *a = a << k*
a >>= k	аналог оператора *a = a >> k*
a &= b	аналог оператора *a = a & b*
a |= b	аналог оператора *a = a | b*
a ^= b	аналог оператора *a = a ^ b*

Таблица 4

Символы некоторых однобайтовых констант		
Символ	Код ASCII	Пояснение
\0xHH	0xHH	шестнадцатеричный код (H – цифра от 0 до F)
\0qqq		восьмеричный код (q – цифра от 0 до 7)
\a	0x07	символ bel, генерирующий звук при выводе
\b	0x08	символ backspace, отменяющий предшествующий символ
\f	0x0C	символ Form Feed – перевод страницы
\n	0x0A	символ Line Feed – перевод строки
\r	0x0D	символ Carriage Return – перевод курсора в начало строки
\t	0x09	символ Tab – горизонтальная табуляция
\v	0x0B	символ вертикальной табуляции
\\	0x5C	символ \ – обратная наклонная черта (обратный слэш)
\’	0x27	символ одинарной кавычки
\″	0x22	символ двойной кавычки
\?	0x3F	символ знака вопроса

Вообще говоря, к «символам» алфавита причисляют и набор служебных слов, используемых в оформлении программы. Ядро Си насчитывает порядка 40 таких слов (см. табл. 5), а с появлением расширения языка в виде C++ к ним добавилось еще порядка 20 новых терминов. С большинством из них мы будем знакомиться по мере освоения материала последующих разделов.

Таблица 5

Служебные слова ядра входного языка Borland C	
Служебное слово	Назначение
Asm	включение команд на языке ассемблера
Auto	редко используемое объявление локальных переменных
Break	досрочный выход из циклов и переключателей
Case	начало строки выбора в переключателе
Cdecl	объявление Си-стиля для функций и данных (различие больших и малых букв, добавление лидирующих подчерков, запись в стек аргументов функций с конца)
Char	объявление однобайтовых символьных или числовых данных
Const	объявление не модифицируемых переменных – именованных констант или параметров, передаваемых по адресу
continue	переход на продолжение цикла
default	строка выбора в переключателе, на которую переходят в случае, если не выбрана ни одна предыдущая строка
do	оператор цикла с постусловием
double	объявление вещественных данных с удвоенной точностью
else	начало альтернативной ветки в условном операторе if
enum	объявление перечислимых данных

extern	ссылка на глобальные данные, описанные в других файлах
far	объявление дальних указателей
float	объявление вещественных данных с обычной точностью
for	оператор цикла
goto	оператор безусловного перехода
huge	объявление дальних указателей
if	условный оператор
int	объявление целочисленных двухбайтовых данных
long	объявление целочисленных или вещественных данных повышенной длины
near	объявление ближних указателей
Pascal	объявление стиля Pascal для функций и данных (большие и малые буквы не различаются, параметры в стек заносятся с начала)
register	указание о размещении переменных в регистровой памяти
return	возврат из функции
short	объявление целочисленных двухбайтовых данных
signed	объявление целочисленных данных со знаком
sizeof	определение длины данных
static	объявление статических локальных переменных, сохраняющих свои значения после выхода из функции
struct	объявление данных типа структура
switch	оператор выбора – переключатель
typedef	переименование стандартного типа данных
union	объявление данных типа объединение
unsigned	объявление целочисленных данных без знака
void	указание об отсутствии аргументов или значения, возвращаемого функцией
volatile	указание о том, что значение переменной может быть изменено другой программой
while	описание условия прекращения цикла

В программах на языке Си можно встретить операцию, в которой участвуют одновременно 3 операнда :

```
min = (a < b) ? a : b ;
```

Символом этой операции является знак вопроса, а выполняется она следующим образом. Если условие, заключенное в скобках выполнено, то значение присваиваемого выражения задается выражением, расположенным справа от знака вопроса. В противном случае вычисляется значение выражения, расположенного справа от двоеточия.

4.2. Форматы основных операторов

Большинство служебных слов языка Си используется для описания данных (объявления) и программных конструкций (операторов), выполняющих определенные действия. С объявлениями мы будем знакомиться по мере освоения различных типов данных. Каждый оператор языка Си завершается точкой с запятой.

Формат основного действия вычислительного характера – оператора присваивания – приведен в табл.3. Его назначение – вычисление значения выражения, расположенного справа от знака присваивания и запись этого значения в область памяти, выделенную для хранения переменной, имя которой указано слева от знака присваивания. Как правило, тип переменной и тип вычисленного значения должны совпадать. Однако, если это не так, то программист может ввести явное указание типа, к формату машинного представления которого должно быть преобразовано значение выражения:

```
v = (тип_переменной_v) e;
```

Если компилятор обнаруживает несоответствие между типами данных, то он выдает соответствующее предупреждение. И хотя оно не рассматривается как ошибка, приводящая к аварийному завершению этапа трансляции, пользователь должен обращать внимание на такого рода предупреждения.

Оператор ветвления, иногда называемый условным, имеет два формата: укороченный – с одной исполнительной веткой, и полный – с двумя альтернативными ветками:

```
if (условие) S1;
if (условие) S1; else S2;
```

Действие *S1* выполняется в том случае, когда проверяемое условие оказывается справедливым (истинным). Если в качестве условия выступает одна из операций отношения, то определение его истинности сомнения не вызывает. Но в программах на языке Си в качестве условия зачастую используется

значение какой-либо переменной или выражения. В этом случае условие считается выполненным, если проверяемое значение отлично от нуля.

Альтернативное действие *S2* выполняется в том случае, если проверяемое условие нарушено.

В большинстве случаев исполнительные ветки условного оператора должны содержать несколько операторов языка. Тогда их приходится заключать в фигурные скобки:

```
if (условие)
   { t1; t2; ...}
else
   { f1; f2;...}
```

Операторы, заключенные в фигурные скобки, образуют один составной оператор.

Обобщением условного оператора является оператор выбора или переключатель. Он вычисляет значение некоторой переменной или выражения и, в зависимости от полученного результата, обращается к выполнению одной из нескольких альтернативных ветвей:

```
switch (переключающее_выражение)
  {
    case  c1:       s11; s12; ... ; break;
    case  c2:       s21; s22; ...; break;
  ..............................................
    case  ck:       sk1; sk2; ...; break;
    default:        s1; s2; .....
  }
```

Группа операторов $s_{11}; s_{12};$ ... выполняется в том случае, если вычисленное значение переключающего выражения совпадает с константой c_1. Оператор *break*, завершающий группу, предотвращает переход на выполнение последующих альтернативных действий и приводит к выходу из оператора *switch*. Если значение переключающего выражения совпадает с константой c_2, то выполняется последовательность действий, представленная операторами $s_{21}, s_{22}, \ldots$. Если переключающее выражение принимает значение, отличное от предусмотренных констант $c_1, c_2, \ldots, c_k$, то выполняется группа операторов по умолчанию – $s_1, s_2, \ldots$. Последняя альтернативная ветвь (*default* ...) в операторе *switch* может отсутствовать.

Операторы цикла предназначены для многократного выполнения участка программы, называемого телом цикла. Количество повторений при этом может быть задано явным образом или определяться условием, которое проверяется либо перед телом цикла (цикл с предусловием), либо в конце тела цикла (цикл с постусловием). Наиболее универсальная конструкция цикла представлена оператором *for*:

```
for  (f1,f2,...; условие; a1,a2,...)
        { тело цикла }
```

Действия *f1, f2,...* выполняются перед входом в цикл. После них проверяется условие, в случае истинности которого выполняется тело цикла. После очередного исполнения тела цикла выполняются действия *a1, a2, ...* и происходит возврат на проверку условия. Как только условие оказывается нарушенным, управление передается оператору, следующему за телом цикла. Это может произойти и при начальном входе в цикл, так что тело цикла может вообще не выполняться. Любая из трех конструкций в заголовке цикла *for* может отсутствовать. Так, например, с помощью оператора *for(;;);* можно организовать бесконечный цикл.

В отличие от других алгоритмических языков, где оператор *for* предусматривает изменение некоторой переменной по закону арифметической прогрессии, Си позволяет организовать цикл с произвольным изменением одной или нескольких переменных. Например:

```
for(s=0, i=1; i < n; s += a[i], i *= 2);
```

Этот оператор определяет сумму элементов массива *a[1] + a[2] + a[4] + a[8] + ...*

В теле любого цикла имеется возможность организовать досрочный выход из цикла с помощью оператора *break*. Еще одно нарушение логики работы тела цикла вызывается оператором *continue*. Он организует переход на продолжение цикла, т.е. либо на группу действий *a1, a2,...* в операторе *for*, либо на проверку условия выхода из цикла в операторах *while* и *do*.

Цикл с предусловием выглядит следующим образом:

```
while (условие)
{тело цикла}
```

Тело цикла выполнятся в случае истинности условия и обходится, если условие нарушено. Здесь также возможна ситуация, когда тело цикла ни разу не выполняется. Перед входом в цикл WHILE в первый раз обычно инициализируют одну или несколько переменных для того, чтобы условное выражение имело какое-либо значение. Оператор или группа операторов, составляющих тело цикла, должны, как правило, изменять значения одной или нескольких переменных, входящих в условное выражение, с тем чтобы в конце концов выражение обратилось в нуль и цикл завершился.

Предостережение. Потенциальной ошибкой при программировании цикла WHILE, как, впрочем, и цикла любого другого типа, является запись такого условного выражения, которое никогда не прекратит выполнение цикла. Такой цикл называют бесконечным.

Пример программы с зацикливанием

```
main()
{
  int i=4;
  while (i > 0)
          printf("\n Турбо Си лучший язык");
}
```

Цикл WHILE завершается в следующих случаях:

- обратилось в ложь условное выражение в заголовке цикла;
- в теле цикла встретился оператор break;
- в теле цикла встретился оператор return;

В первых двух случаях управление передается на оператор, располагающийся непосредственно за циклом, в третьем случае - происходит выход из функции.

Еще одна распространенная ошибка, приводящая к возникновению бесконечного цикла. Ошибка состоит в том, что при написании условного выражения вместо оператора сравнения на равенство (==) используется оператор присваивания (=). Компилятор Borland C выдает предупреждение о возможной ошибке.

Следует обращать внимание на все предупреждения, выдаваемые компилятором. В конце концов вы обнаружите причину появления такого сообщения и, возможно, исправите что-то в программе.

Формат оператора цикла с постусловием таков:

```
do
{тело цикла}
while (условие);
```

И здесь тело цикла продолжает повторяться до тех пор, пока сохраняется истинность условия. Но, в отличие от двух предыдущих циклов, тело цикла с постусловием всегда выполняется хотя бы один раз.

Цикл DO WHILE завершается в тех же случаях, что и цикл WHILE.

Перед любым исполняемым оператором программы может быть расположена символьная или числовая метка, отделяемая от оператора двоеточием. На помеченный таким образом оператор можно передать управление с помощью оператора безусловного перехода:

```
goto m;
```

При этом следует помнить, что помеченный оператор и оператор *goto* должны находиться в одной и той же функции. Переход из одной функции в другую с помощью оператора *goto* недопустим. Вообще говоря, безусловными переходами следует пользоваться в исключительных случаях.

Любая программа на Си оформляется в виде одного или нескольких программных модулей – функций. Функция, с которой начинается выполнение программы, должна иметь стандартное название – *main*. Функция может иметь или не иметь аргументы, но даже в последнем случае при обращении к функции после ее имени должны указываться пустые скобки. Например:

```
clrscr();
```

Для возврата из тела вызываемой функции используется оператор *return*. Если вызываемая функция должна возвратить значение, то оно указывается вслед за служебным словом *return*. Например, функция, определяющая знак целочисленного аргумента, может быть оформлена следующим образом:

```
int sign(int n)
{
        if (x < 0) return –1;
        if (x > 0) return 1;
        return 0;
}
```

К числу операторов описания данных условно можно отнести конструкцию подстановки, начинающуюся со служебного слова *#define*:

#define $a_1a_2...a_k$ $b_1b_2...b_n$

Ее смысл заключается в том, что перед компиляцией в тексте исходной программы производится замена последовательности символов $a_1a_2...a_k$ на цепочку $b_1b_2...b_n$. Такая замена может оказаться полезной в двух ситуациях. Во-первых, она позволяет определять константы, значения которых могут варьироваться при компиляции очередного варианта программы. Во-вторых, таким образом могут делаться макроподстановки, напоминающие встроенные внутренние функции с настраиваемыми фрагментами. Например, макроопределение функции, вычисляющей максимум из двух аргументов, может выглядеть следующим образом:

```
#define max((a),(b))  ((a) > ((b))? (a) : (b)
```

Казалось бы, что в приведенном примере слишком много скобок. Однако они позволяют подставить вместо аргументов *a* и *b* любые выражения. Иногда текст подставляемой функции может оказаться достаточно длинным, не помещающимся в одной строке. Тогда макроопределение можно перенести на следующую строку, поставив в конце символ переноса – обратный слэш (\ ). В отличие от большинства операторов строки подстановок не завершаются точками с запятой. Исключение может составить макропроцедура, тело которой должно завершаться таким разделителем.

Весьма полезно сопровождать программу комментариями, поясняющими назначение переменных и смысл наиболее сложных фрагментов программы. Borland C допускает два варианта включения комментариев. В первом случае комментарий начинается после двух подряд идущих символов // и продолжается до конца строки. Во втором случае комментарий может быть и многострочным. Его начинает пара символов /* и завершает такая же пара, записанная в обратном порядке – */.

4.3 Структура простых программ на Си

Рассмотрим задачу, которая должна дать ответ на вопрос, является ли данное число N простым или составным, т.е. имеет ли оно какие-либо делители, кроме 1 и N, или нет.

Один из наиболее легко реализуемых алгоритмов проверки числа на "простоту" заключается в том, исходное число N последовательно делят на 2, 3, 5, 7, 9, ..., 2*p+1 ($(2*p+1)^2 \le N$). Если ни один из остатков от деления не равен нулю, то N – простое. Конечно, этот алгоритм далек от совершенства – например, зачем делить на 9 или 15, если число не делилось на 3. По идее, в проверке должны бы участвовать только простые делители - 2, 3, 5, 7, 11, 13,... Но построить такую программу гораздо сложнее.

Приведенная ниже программа, реализующая предложенный алгоритм, содержит основные компоненты любой программы на языке Си. Для пояснения назначения и действия отдельных ее строк использована нумерация, отсутствующая в реальной программе.

```
01.   #include <stdio.h>
02.   #include <conio.h>
03.   main()
04.   {
05.         long N,j;
06.         printf("\nВведите целое число: ");
07.         scanf("%ld",&N);
08.         if(N < 4)
09.         {
10.               printf("\nЭто число - простое");
11.               getch();
12.               exit(0);
13.         }
14.         if(N % 2 == 0)
15.         {
16.               printf("\nЭто число - составное");
17.               getch();
18.               exit(0);
19.         }
20.         for(j = 3; j*j <= N; j += 2)
21.         if( N % j == 0)
22.         {
23.               printf("\nЭто число - составное");
24.               getch();
25.               exit(0);
26.         }
27.         printf("\nЭто число - простое");
28.         getch();
29.         return;
30.   }
```

Две первые строки программы подключают к тексту программы так называемые заголовочные файлы системы с именами *stdio.h* и *conio.h*. В этих файлах описаны системные функции (в языке Си любые процедуры - подпрограммы и функции, - принято называть функциями), обеспечивающие стандартные операции ввода/вывода (standard input/output) и взаимодействия с консолью (console input/output). Смысл

заголовочных файлов заключается в проверке правильности обращений к системным функциям, которые содержатся в программе.

Строка с номером 03 содержит заголовок головной (главной) функции, которая в любой программе на Си должна иметь стандартное имя *main*. Пустые скобки после имени функции означают, что этой функции не передаются параметры. Иногда можно встретить и другую запись, имеющую точно такой же смысл - *main(void)*. Служебное слово *void* означает в данном случае отсутствие аргументов у функции.

Тело любой программы, следующее за ее заголовком, на языке Си заключается в фигурные скобки, играющие такую же роль, например, как операторные скобки *begin - end* в языке Паскаль. В нашем примере тело программы открывает фигурная скобка в строке 04, а соответствующая ей закрывающаяся скобка находится в строке 30.

Строка 05 содержит описание переменных *N* и *j*, используемых в тексте программы, с указанием их типа *long*. Это означает, что для каждой переменной в оперативной памяти машины будет отведено по 4 байта и значения таких переменных могут быть только целочисленными.

Строка 06 заставляет компьютер вывести на экран предложение "Введите целое число: ", которое появится в начале строки. Указание \n на Си воспринимается как приказ перейти в начало следующей строки.

В ответ на приглашение компьютера человек должен набрать на клавиатуре число, подлежащее анализу на простоту, и нажать клавишу Enter. Набранная информация принимается функцией *scanf* (строка 07) и направляется в память по адресу, занимаемому переменной *N* (запись вида *&N* читается на Си как "адрес переменной N"). Указание *%ld*, предшествующее адресу переменной, означает, что введенное число должно принадлежать диапазону данных, соответствующих типу *long*.

Во избежание лишних проверок все числа, не превосходящие 4, объявляются простыми. И такая проверка выполняется в строке 08. Если условие *N<4* справедливо, то выполняются действия строк 10-12, заключенные в фигурные скобки 09 и 13. Строка 10 выводит сообщение о том, что введенное число является простым. Для того, чтобы приостановить работу программы и рассмотреть выданное сообщение, использовано обращение к системной функции *getch* (строка 11), которая будет ждать до тех пор, пока пользователь не нажмет какую-либо клавишу. Последующее обращение к функции *exit* приводит к прекращению работы программы.

Если условие *N<4* не выполнено, то действия в фигурных скобках 09-13 обходятся и программа проверяет следующее условие, заключающееся в том, делится ли *N* на 2 без остатка. Операция, обозначаемая в Си символом процента (%), определяет остаток от деления первого операнда на второй. Использованный далее немного необычный знак в виде двух символов "равно" (= =) в Си означает проверку на равенство. Одиночный знак "равно" используется в Си только как знак оператора присваивания.

Если условие строки 14 выполнено, что соответствует четности числа *N* (а этой проверке предшествовало не выполнившееся условие *N<4*), то выполняются действия в фигурных скобках 15-19, выдающие на экран сообщение о том, что число *N* является составным. В противном случае эти действия обходятся.

Строка 20 представляет собой заголовок цикла в котором переменная *j* последовательно принимает нечетные значения от 3 до максимально проверяемого делителя ($j^2 \le N$). Заголовок цикла начинается со служебного слова *for* (для), после которого в круглых скобках записывается три конструкции, разделенные точками с запятой и определяющие соответственно действие перед входом в цикл (*j=3*), условие окончания цикла (*j*j <= N*) и действие после выполнения тела цикла (запись *j += 2* эквивалентна оператору присваивания *j=j+2*).

Собственно тело цикла представлено единственным оператором проверки условия, делится ли *N* на *j* без остатка (строка 21). Если это условие выполнено, то программа обнаружила делитель, свидетельствующий о том, что число *N* составное. И в этом случае срабатывают строки, заключенные в фигурных скобках 22-26. Строка 23 выведет на экран соответствующее сообщение, строка 24 приостановит выполнение программы, а строка 25 прервет выполнение программы.

Если во время работы цикла не будет обнаружен ни один делитель числа *N*, то программа переходит к строке 27, где организован вывод сообщения о том, что число *N* является простым.

Приведенный текст программы далек от совершенства, в нем наблюдаются многочисленные повторы. Для сравнения мы приведем второй вариант, в котором анализ типа числа вынесен в отдельную функцию *prime*, возвращающую значение 1, если ее аргумент является простым числом, и 0 - в противном случае.

```
#include <stdio.h>
#include (conio.h>
int prime(long N);
main()
{
```

```
long M;
char *a[]={"составное","простое"};
printf("\nВведите целое число: ");
scanf("%ld",&M);
printf("\nЭто число - %s",a[prime(M)]);
getch();
}
int prime(long N)
{
long j;
if(N < 4) return 1;
if(N % 2 == 0) return 0;
for(j = 3; j*j <= N; j += 2)
if( N % j == 0) return 0;
return 1;
}
```

В этом примере в дополнительных пояснениях нуждаются только несколько новых строк. Третья строка содержит так называемый прототип функции *prime*, напоминающий ее заголовок и отличающийся от последнего только наличием точки с запятой в конце строки. Прототип дает возможность компилятору проверить правильность обращения к функции на соответствие типов и количества передаваемых параметров.

Во-вторых, в седьмой строке описан символьный массив *a* (точнее, массив из двух указателей символьного типа), содержащий две строки со словами "простое" и "составное". Выбирая элемент *a[0]* из этого массива, мы получим адрес строки с текстом "простое". Второй элемент этого массива *a[1]* "смотрит" на слово "составное".

Наконец, оператор *return*, использованный для возврата из функции *prime* имеет аргумент – 0 или 1, который рассматривается как значение функции после ее работы.

Второй пример наглядно демонстрирует улучшение структуры программы за счет разумного разбиения ее на автономные фрагменты. Создаваемые таким образом функции могут быть использованы и при решении других задач.

Известно, что любое четное число *N (N > 0)* может быть представлено в виде суммы двух простых чисел *(N = N1 + N2)*. Ниже приводится текст программы, решающей такую задачу. Она отыскивает все такие разложения (о не единственности решения свидетельствует простейший пример: (4 = 1 + 3 = 2 + 2). Алгоритм работы программы несложен. Она перебирает все возможные слагаемые, первое из которых не превосходит *0.5*N*, и если оба они оказываются простыми, выводит полученный результат. Анализ слагаемых выполняется с помощью ранее составленной функции *prime*.

```
#include <stdio.h>
#include <conio.h>
int prime(long N);
main()
{
long M,j;
clrscr();
printf("\nВведите четное число: ");
scanf("%ld",&M);
if(prime(M-2))  printf("%ld=  2+%ld\t",M,M-2);
for(j=1; j <= M/2; j+=2)
if(prime(j) && prime(M-j))  printf("%ld=%3ld+%ld\t", M,j,M-j0);
getch();
}
int prime(long N)
{
long j;
if(N < 4) return 1;
if(N%2 == 0) return 0;
for(j=3; j*j <= N; j+=2)
if(N%j==0) return 0;
return 1;
}
```

В новой программе в дополнительных пояснениях нуждаются только два оператора. В 10 строке делается попытка представить введенное число в виде суммы *2+(N-2)*. Для этого проверяется условие *prime(N-2)=1*. Но в операторе *if* в явном виде отсутствует условие *prime(N-2)==1*. Дело в том, что Си рассматривает любое число, отличное от 0, как истину. Поэтому проверки *if(prime(N-2))* и *if(prime(N-2)==1)* дают одинаковый результат.

Еще один элемент новизны содержится в операторе вывода результата, где нужно выдать на экран сообщение вида *N=a+b*. Для каждого из трех чисел, имеющих тип *long* в операторе *printf* присутствует указатель вида *%ld*. Символы "=" и "+" вставляются в текст выводимого сообщения после соответствующего числа. Указание \t соответствует переводу курсора в новую колонку (табуляторный пропуск) для последующего вывода нового разложения, если таковое будет обнаружено.

4.4 Работа с числовыми данными

Современные представления о числовых данных базируются на так называемых позиционных системах счисления. Для этих систем характерно основание системы p, степень которого определяет вес цифры a (a=0,1,2,...,p-1) в зависимости от занимаемой позиции:

$$a_k\, a_{k-1} \ldots a_1\, a_0, b_{-1}\, b_{-2} \ldots b_{-m} = a_k*p^k + a_{k-1}*p^{k-1} + \ldots + a_1*p^1 + a_0*p^0 + b_{-1}*p^{-1} + b_{-2}*p^{-2} + \ldots + b_{-m}*p^{-m}$$

Главенствующую роль в человеческом общении играет десятичная система счисления (p = 10; a = 0,1,...,9). Однако ЭВМ используют более рациональную двоичную систему (p = 2; a = 0,1). Единственным ее недостатком является большая длина чисел. Количество разрядов (цифр) в двоичном представлении числа примерно в 3 раза превышает количество десятичных цифр. И для преодоления этого неудобства программисты прибегают к более компактной записи двоичных кодов в виде восьмеричных или шестнадцатеричных чисел. При этом три или четыре смежные двоичные цифры заменяют одной восьмеричной или шестнадцатеричной цифрой. Например:

$192_{10} = 11000000_2$

$11000000_2 = 300_8$

$11000000_2 = C0_{16}$

4.4.1. Внешнее и внутреннее представление числовых данных

Под внешним представлением числовой информации мы подразумеваем способы записи данных, используемые в текстах программ, при наборе чисел, вводимых в ЭВМ по запросу программы, при отображении результатов на экране дисплея или на принтере. Кроме естественного представления числовых констант в виде целого или вещественного числа языки программирования допускают различные добавки в начале ("префиксы") или конце ("суффиксы") числа, определяющие способы преобразования и хранения данных в памяти компьютера.

В Си к таким суффиксам относятся указания об удвоенной длине целых чисел (буквы *L* или *l*), указания о вещественном формате числа, не содержащего в своей записи десятичной точки или десятичного порядка (буква *F* или *f*), указания об использовании без знакового представления целых чисел (буква *U* или *u*). Префиксы в Си используются для записи восьмеричных (число начинается с *0*) или шестнадцатеричных (числу предшествует одна из комбинаций *0x* или *0X*) констант:

5 – короткое целое число со знаком;
5U – короткое целое число без знака;
5L – длинное целое число со знаком;
5LU или *5UL* – длинное целое число без знака;
05 – восьмеричное число;
0x5 – шестнадцатеричное число;
5f – вещественное число со знаком.

Наличие в естественной записи числа точки (3.1415) или указателя десятичного порядка (314.159265e-02) означает, что соответствующее значение представлено в ЭВМ в виде вещественного числа с плавающей запятой.

Машинные форматы представления чисел мы будем называть внутренними и из приведенных ранее примеров следует, что машинные числа бывают целыми и вещественными. В свою очередь, каждый из этих типов данных допускает несколько представлений, отличающихся диапазоном допустимых чисел. Остановимся более подробно на машинных форматах числовых данных, соответствующих их аналогам в алгоритмических языках.

Самые короткие числа со знаком представлены в памяти ЭВМ одним байтом, в котором может разместиться любое число из диапазона от -128 до 127. В Си для описания данных такого типа используется спецификатор *char*.

В одном же байте может быть расположено и самое короткое целое число без знака. В Си для описания таких данных служит спецификатор *unsigned char*. Диапазон допустимых данных при этом смещается вправо и равен [0, 255].

Вторая категория целых чисел представлена двухбайтовыми данными. В варианте со знаком они предлагают диапазон от -32768 до +32767, в варианте без знака - от 0 до 65535.

Си использует для описания двухбайтовых целочисленных данных спецификаторы *int* и *unsigned int*, корректно выполняя арифметические операции и с без знаковыми данными при условии, что результат не выходит за пределы разрешенного диапазона.

Третья категория целых чисел в IBM PC представлена четырехбайтовыми данными. В варианте со знаком они перекрывают диапазон от -2147483648 до +2147483647, в варианте без знака - от 0 до 4294967295.

Для описания четырехбайтовых данных целого типа в Си используются спецификаторы *long* (эквивалент *long int*) и *unsigned long*.

Следует помнить, что для хранения любых целых чисел со знаком в IBM PC используется дополнительный код, что сказывается на представлении отрицательных чисел:

+5 = 0 0000101 = 0 000000000000101

–5 = 1 1111011 = 1 111111111111011

Наиболее часто применяемые типы вещественных чисел представлены короткими (4 байта) и длинными (8 байт) данными. Си использует для этой цели спецификаторы *float* и *double*.

Короткий вещественный формат по модулю обеспечивает представление чисел в диапазоне от 10^{-38} до 10^{+38} примерно с 7-8 значащими цифрами. Для 8-байтового формата диапазон существенно расширяется –от 10^{-308} до 10^{+308}, а количество значащих цифр увеличивается до 15-16.

Сопроцессор IBM PC предлагает еще два формата данных, занимающих соответственно 8 и 10 байт. Первый из них допускает работу с целыми числами из диапазона от -2^{63} до $2^{63}-1$. Второй формат перекрывает диапазон данных (по модулю) от 10^{-4932} до 10^{+4932}, сохраняя 19-20 значащих цифр. Расширенный формат вещественных данных можно использовать и в программах на Си (*long double*).

В машинном представлении вещественных данных разного типа на IBM PC не выдержана какая-то общая идеология. Объясняется это, по всей вероятности, разными наслоениями на прежние аппаратные решения, которые принимались при разработке процессоров в разных отделениях фирмы Intel. Поэтому здесь имеют место такие нюансы как сохранение или не сохранение старшего бита мантиссы, представление мантиссы в виде чисто дробного ($0.5 \leq m < 1$) или смешанного ($1 \leq m < 2$) числа и т.п. Прикладных программистов эти детали мало интересуют, однако при создании специальных системных компонент с точным представлением данных приходится считаться.

4.4.2. Ввод числовой информации

Каких-либо особых проблем с вводом числовой информации в программах не возникает. В Си основные неприятности форматного ввода (функция *scanf*) связаны с попыткой указать в списке вода не адрес переменной, а ее имя:

```
scanf("%d",x);              //правильно было бы  scanf("%d",&x);
```

Компилятор TC/BC такую ошибку, к сожалению, не замечает и преобразует имя *x* в какой-то фантастический адрес. Последующую работу программы в этом случае предсказать трудно. Не обращает внимания компилятор Си и на несоответствие между спецификатором формата и типом переменной из списка ввода. Всего этого можно избежать используя потоковый ввод:

```
cin >> x;
```

4.4.3. Вывод числовых результатов

Наиболее приятный вид имеет числовая информация, организованная по табличному типу – в виде колонок фиксированной ширины, в которых одноименные числовые разряды располагаются друг под другом (единицы под единицами, десятки под десятками сотни под сотнями и т.д.). При этом, в частности, более рационально используется площадь экрана. Достигается такая красота за счет управления форматами выводимых данных.

В Си для этой цели используется функция printf:

```
printf("A=%6.3f B=%3d C=%10.4e",A,B,C);
```

При выводе в поток, когда к Си-программе подключаются заголовочные файлы *iostream.h* и *iomanip.h*, тоже имеется возможность управлять форматом выводимых данных:

```
cout << "A=" << setw(6) << setprecision(5) << A;
```

Среди форматных спецификаторов в Си есть дополнительные возможности, обеспечивающие вывод числовых данных не только в десятичной системе:

printf("%4x %6o",x,y);

Приведенные здесь спецификаторы позволяют вывести значения целочисленных переменных *x* и *y*, соответственно, в виде шестнадцатеричного числа с четырьмя цифрами и восьмеричного числа с шестью цифрами.

Си позволяет прижимать выводимое число к левой границе отведенного поля (по умолчанию действует правый прижим), печатать знак "+" у положительных чисел, подавлять или разрешать вывод незначащих нулей и др. Подробную информацию о структуре форматного описателя можно найти в файлах помощи системы Borland C++.

4.5. Обработка текстовой информации

4.5.1. Символьные данные и их внутреннее представление

Символьная (текстовая) информация - самый простой тип данных с точки зрения его представления в памяти ЭВМ. Каждому символу текста в памяти соответствует байт с 8-разрядным кодом этого символа в том или ином стандарте. Буквам латинского алфавита, цифрам, знакам операций и различным разделителям (скобки, точки, запятые и т.п.) в некотором смысле повезло больше, так как их кодировка, практически, универсальна. Она предложена фирмой IBM и составляет первую половину большинства 8-разрядных кодировочных таблиц, используемых в разных странах. В терминологии IBM PC такие таблицы принято называть кодовыми страницами. Вторая половина кодовых страниц отведена под национальную символику.

В отечественной практике чаще всего приходится иметь дело либо с символикой MS-DOS в стандарте ASCII (American Standard Code for Information Interchange), либо с одной из кодовых страниц ANSI (American National Standard Institute), применяемых в среде Windows. Между этими таблицами существует принципиальное различие, связанное с кодировкой букв русского алфавита.

В таблице ASCII (кодовая страница 866) заглавные русские буквы начинаются со 128-й позиции. Вплотную вслед за ними располагается цепочка малых букв от *"а"* (код - 160) до буквы *"п"* (код - 175). Далее следуют символы псевдографики, используемые для формирования таблиц с одинарными и двойными линиями (диапазон кодов от 176 до 223). Начиная с 224-й позиции располагаются остальные буквы от *"р"* до *"я"*. И, наконец, вслед за ними, выбиваясь алфавитного порядка, размещены буквы *"Ё"* (код – 240) и *"ё"* (код – 241).

В таблице ANSI (кодовая страница 1251) коды русских букв начинаются с позиции 192 (код буквы *"А"*) и расположены сплошным интервалом до 255-й позиции (код буквы *"я"*). Символы псевдографики в графической оболочке Windows смысла не имеют и поэтому в таблице ANSI отсутствуют. Буквы *"Ё"* и *"ё"* здесь тоже не включены в общепринятую алфавитную последовательность и имеют, соответственно коды 168 и 184. Более того, их обозначение на большинстве русифицированных клавиатур отсутствует и для включения таких букв в набираемый текст приходится нажимать самую левую клавишу в верхнем ряду (на латинском регистре этой клавише соответствует символ "~").

Все достаточно развитые алгоритмические языки включают серию процедур по обработке символьных (каждый объект – отдельный символ) и строковых (цепочки символов) данных. Базовым текстовым элементом в Си являются объекты типа *char*, значением каждого их которых является единственный символ. Объединение таких объектов в одномерный массив позволяет работать как с цепочками символов, так и с отдельными символами. В Си это вытекает из способа представления текстовой информации. Начальное присвоение (*char name[5]="Вася";*) или копирование одной строки в другую (*strcpy(name,"Вася");*) эквивалентно 5 обычным операторам присваивания:

```
name[0]='В';
name[1]='а';
name[2]='с';
name[3]='я';
name[4]='\0';
```

С точки зрения внутреннего представления текстовых данных в памяти ЭВМ в Си вслед за цепочкой символов располагается байт с признаком конца текста (обычно – это нулевой код).

4.5.2. Ввод и вывод текстовой информации

В Си имеется довольно много возможностей для ввода одиночных символов и цепочек строк. Форматный ввод с помощью функции *scanf* использует для этой цели два спецификатора:

"%s" – при вводе строковых значений в массив типа *char*;

"%c" – при вводе одиночных символов в переменную типа *char*.

Например:

```
char c1, s1[80];
```

```
scanf("%c", &c1);          //не забывайте указывать адрес переменной
scanf("%s", s1);           // имя массива одновременно является адресом
```

Ввод символьных и строковых данных завершается после нажатия клавиши Enter, однако в переменную *c1* поступит только один символ, а в массив *s1* при этом будет введена начальная цепочка символов до первого пробела. В операторе форматного ввода пробел или несколько подряд идущих пробелов рассматриваются как признак конца информации, считываемой из очередного поля. Конечно, при наборе вы можете набрать в одной строке несколько символов или несколько слов, разделенных пробелами. Все они будут введены в системный буфер, выделяемый для работы функции *scanf* (стандартный размер этого буфера позволяет набрать строку длиной до 128 символов, включая и разделители-пробелы). При последующих обращениях к функции ввода данные из буфера продолжают извлекаться до тех пор, пока не будут исчерпаны, после чего вновь придется продолжить набор на клавиатуре.

Конечно, такой способ ввода может доставить неприятности и привести к непредусмотренному продолжению работы программы. Понятно также, что для ввода одного символа не хочется нажимать две клавиши, а вводимые строки могут состоять и из нескольких слов. Поэтому ввод символьных данных лучше выполнять с помощью других процедур:

```
int c1;
c1 = getch();              //Ввод кода нажатой клавиши без
//отображения соответствующего символа на экране
c1 = getche();             //Ввод кода нажатой клавиши с отображением
//соответствующего символа на экране
c1 = getchar();    //Ввод кода нажатой клавиши вслед за
//нажатием клавиши Enter
```

Обратите внимание на то, что вводимый символ передается не в переменную типа *char*, а в двухбайтовую целочисленную переменную. Именно такие значения возвращают указанные выше функции.

Вторая особенность их применения связана с разбиением клавиатуры на две категории клавиш – отображаемые и управляющие. Окраска клавиш не совсем точно передает принадлежность клавиши той или иной группе. Например, функциональные клавиши F1, F2, ..., F12 имеют разный цвет, но все они относятся к категории управляющих. А клавиша Enter, несмотря на серую окраску причислена к разряду обычных.

Дело в том, что при нажатии обычной клавиши в буфер входного потока (*stdin*) поступает единственный байт с ненулевым кодом. Именно он и извлекается при первом же обращении к одной из функций *getch, getchar* или *getche*. От нажатия управляющих клавиш в буфер *stdin* поступают 2 байта, первый из которых содержит нулевой код, а второй представляет уже собственно числовой код, соответствующий выбранной клавише. Для извлечения второго байта к этим функциям приходится обращаться повторно и таким образом программа может проанализировать сложившуюся ситуацию.

Есть некоторые нюансы и при нажатии клавиши Enter. Так, функция *getch* сообщает, что нажата клавиша с числовым кодом 13, а функция *getchar* считает, что последним введен символ с кодом 10. На самом деле она перекодирует код клавиши Enter в символ LF (*line feed* – строка заполнена), действительно имеющий код 10 (0x0A) в таблице ASCII. По разному воспринимают эти функции и комбинацию Ctrl+z, сообщая в одном случае, что поступил признак конца файла (*end-of-file*) с кодом 26, а в другом - что встретился признак EOF, которому соответствует числовой код -1. Функция *getchar* не позволяет ввести код клавиши Esc, тогда как функции *getch* и *getche* успешно справляются с этой клавишей.

Зато с функций *gets*, осуществляющей ввод строки, у вас никаких проблем не возникнет:

```
gets(s);
```

Вы можете набирать любые предложения, содержащие любое количество пробелов, и все они будут введены в строку *s*. Однако длина вводимой строки ограничена емкостью буфера (128 байт).

Дополнительные возможности по вводу текстовых данных связаны с использованием потоков:

```
#include <iostream.h>
.....
char c1, s1[80];
.....
cin >> c1;
cin >> s1;
```

С выводом символьных и строковых значений все обстоит гораздо проще. Форматный вывод в Си при помощи функции *printf* использует указанные выше спецификаторы, однако константные текстовые данные, которыми перемежаются выводимые значения, здесь располагаются между спецификаторами формата:

```
printf("%c Вася %s",c1,s1);
```

Вывод отдельного символа или одиночной строки в Си можно выполнить и с помощью функций *putchar(c1)* и *puts(s1)*.

Выводимое значение обычно располагается на экране, начиная с текущей позиции курсора. Если необходимо разместить текст в заранее предусмотренном месте, следует прибегнуть к одной из служебных процедур предварительного перемещения курсора в заданную позицию:

gotoxy(col,row);

Параметр *col* задает номер колонки в диапазоне от 1 до 80, а второй аргумент (*row*) определяет номер строки в диапазоне от 1 до 25.

Еще один способ управления по размещению текста связан с заданием ширины поля, отведенного под выводимое значение. Отображаемый текст при этом прижимается к правой границе поля.

В Си ширина поля включается в спецификатор формата ("%3c %10s"). Однако здесь имеется дополнительная возможность указать, что выводимое значение требуется прижать к левой границе выделенного поля ("%-3c %-10s").

4.5.3. Обработка фрагментов строк

Под фрагментом строки понимают цепочку символов заданной длины, выделенную из исходной строки, начиная с указанной позиции. В частности, выделяемый фрагмент может состоять и из единственного символа. Кроме выделения фрагмента тем или иным способом к числу наиболее распространенных операций относятся действия по определению длины строки, объединению (конкатенации) символьных цепочек и поиску вхождения одной строки в другую.

Функции работы со строками в Си включены в состав заголовочного файла *string.h*. Для копирования строки или ее части в другую здесь можно воспользоваться одной из следующих функций:

strcpy(s1,s2); //копирует строку *s2* в строку *s1*

strncpy(s1,s2,n); //копирует первые *n* символов из строки *s2* в *s1*

Задавая аргумент-источник не ссылкой на начало символьного массива, *а* адресом любого его элемента, мы можем скопировать либо правую, либо среднюю подстроку :

strcpy(s1,&s2[k]); //копирует правую подстроку из *s2* в *s1*

strncpy(s1,&s[2],n); //копирует среднюю подстроку из *s2* в *s1*

Длина строк в среде программирования Borland C++ определяется системной функцией *strlen*. Единственным ее аргументом является анализируемая строка.

В Си операция конкатенации (объединения) строк реализуется с помощью одной из следующих функций:

strcat(s1,s2); //добавляет *s2* к *s1*

strncat(s1,s2,n); //добавляет *n* первых символов из *s2* к *s1*

Поиск вхождения одной строки в другую дает ответ на вопрос, содержится ли значение одного текста в другом и с какой позиции обнаружено это вхождение. Нулевая позиция в качестве результата такой операции соответствует отрицательному ответу.

Си предлагает довольно разнообразные варианты поиска вхождений:

strstr(s1,s2); //ищет вхождение строки *s2* в *s1*

strchr(s1,c); //ищет вхождение символа *c* с начала строки *s1*

strrcgr(s1,c); //ищет вхождение символа *c* с конца строки *s1*

strpbrk(s1,s2); //ищет вхождение любого символа из *s2* в *s1*

strspn(s1,s2); //ищет вхождение любого фрагмента,

//составленного из символов *s2* в *s1*

4.5.4. Сравнение и сортировка текстовых данных

Операции сравнения отдельных символов или строк основаны на последовательном анализе отношений числовых значений соответствующих кодов. В кодовых страницах символы букв упорядочены в соответствии их расположением в латинском или национальном алфавитах. Поэтому код буквы "А" меньше кода буквы "F", код буквы "Г" меньше кода буквы "Ю" и т.д.

Некоторое неудобство вызывает тот факт, что одноименные большие и малые буквы имеют разные коды – в одной клетке кодировочной таблицы можно разместить только один символ, кроме того большие и малые буквы имеют разный смысл. Это не позволяет напрямую упорядочить слова в соответствии с их лексикографическим расположением в словарях. Поэтому приходится предварительно заменять коды всех малых букв в тексте на коды больших (или наоборот) и только после этого выполнять операцию сравнения. Такая замена для букв латинского алфавита особых проблем не представляет, т.к. смещение между кодами соответствующих больших и малых букв - величина постоянная. А вот с русскими буквами приходится повозиться – в кодировке ASCII цепочка малых букв между "п" и "р" разорвана символами псевдографики, а буквы "Ё" и "ё" вообще находятся не на своих местах.

Учитывая эту специфику следует достаточно внимательно использовать языковые средства, связанные с преобразованием или игнорированием разницы в кодировке больших и малых букв. Для русскоязычных текстов их применять нельзя.

Си позволяет преобразовывать содержимое символьных строк к верхнему (*strupr(s)*) или к нижнему (*strlwr(s)*) регистру. Но коды символов, не принадлежащих множеству букв латинского алфавита, остаются при этом без изменения.

Для сравнения строк Си предлагает довольно много системных функций, но не забывайте, что их действие не всегда допустимо над русскими словами. Каждая из описываемых ниже функций принимает положительное значение, если ее первый операнд строго "больше" второго, нулевое значение при "равенстве" операндов, и отрицательное значение, если первый операнд оказался "меньше".

```
strcmp(s1,s2);          //сравнивает строки s1 и s2
strcmpi(s1,s2);         //сравнивает s1 и s2 с игнорированием
//разницы между большими и малыми буквами
stricmp(s1,s2);         //эквивалентна функции strcmpi
strncmp(s1,s2,k); //сравнивает первые k символов в s1 и s2
strncmpi(s1,s2,k);      //сравнивает первые k символов в s1 и s2
//с игнорированием разницы между большими
//и малыми буквами
strnicmp(s1,s2,k);      //эквивалентна функции strncmpi
```

4.5.5. Управление цветом в текстовом режиме

Наиболее важная информация, которую можно почерпнуть из многочисленных источников (книги, руководства по системам программирования, файлы помощи), сводится к трем следующим фактам:

- существует несколько текстовых режимов работы монитора, различающихся по цветовой гамме, а также по числу строк и столбцов, отображаемых на экране;
- содержимое экрана отражает состояние активной страницы видеопамяти;
- каждой символьной позиции экрана (так называемому, знакоместу) в видеопамяти соответствуют 2 байта, в первом из которых находится код ASCII отображаемого символа, а во втором - атрибуты, управляющие цветом пикселов переднего плана (контуры буквы) и фоновых пикселов.

Самый распространенный текстовый режим допускает отображение в цвете 25 строк, каждая из которых содержит по 80 символов. При этом объем соответствующей страницы видеопамяти равен 4000 байт и каждый символ, отображаемый на экране, в принципе, может иметь индивидуальные цветовые характеристики, не зависящие от цветовой гаммы в соседних позициях.

В любом руководстве приводится следующее описание формата байта цветовых атрибутов:

7	6	5	4	3	2	1	0
B	b	b	b	I	f	f	f

Самый старший бит управляет режимом мерцания символа, которое осуществляется аппаратным способом примерно с частотой 1 раз в сек. при *B=1*. Три следующие бита *(bbb)* представляют код цветности фона, который должен принадлежать интервалу [0,7]. Четверка младших битов *(Ifff)* определяет код цветности переднего плана. Среди них особо выделяют бит *I*, единичное значение которого, обычно, соответствует повышенной яркости, т.е. цвету, сопровождаемому приставкой "ярко" или "светло".

Из сказанного выше следует, что стандартный текстовый режим на цветном мониторе позволяет задавать для каждого символа один из 8 фоновых оттенков и один из 16 цветов, выделяющих контур символа. А если мы хотим вывести мерцающий текст, то байты атрибутов соответствующих символов должны быть увеличены на 128.

Управление содержимым байта цветовых атрибутов осуществляется с помощью одной или нескольких системных процедур (операторов), действие которых распространяется на все последующие операции вывода до новой переустановки характеристик цветности.

В Си код цветности символов задается с помощью процедуры *textcolor(fc)*. Значение фонового цвета изменяется процедурой *textbackground(bc)*. В системе программирования Borland C++ существует и более универсальная процедура *textattr*, позволяющая за одно обращение изменить оба цвета и установить признак мерцания:

```
textattr(B*128 + bc*16 + fc);
```

Следует отметить, что при работе с мониторами SVGA сравнительно давно разработанная система Borland C++ 3.1 не очень точно следует ограничениям, вытекающим из структуры байта цветовых

атрибутов. В указанных выше процедурах вы можете задавать цвет фона из диапазона [0,15], а эффект мерцания символов может и не наблюдаться.

Кроме того, в Си для вывода разноцветного текста, следует использовать не стандартную функцию вывода *printf*, а функцию *cprintf*, использующую "прямое" обращение к видеопамяти.

4.6. Функции

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

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

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

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

4.6.1 Основные сведения о функциях

Определение любой функции имеет следующий вид:

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

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

dummy() { }

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

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

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

return *выражение*;

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

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

Функция *main* возвращает в качестве результата число, доступное той среде, из которой данная программа была вызвана.

Механизмы компиляции и загрузки Си-программ, расположенных в нескольких исходных файлах, в разных системах могут различаться. {Здесь еще раз про файл проекта и его создание.}

4.6.2. Функции, возвращающие нецелые значения

В предыдущих примерах функции либо вообще не возвращали результирующих значений (void), либо возвращали значения типа int. А как быть, когда результат функции должен иметь другой тип? Многие вычислительные функции, как, например, sqrt, sin и cos, возвращают значения типа double; другие специальные функции могут выдавать значения еще каких-то типов. Чтобы проиллюстрировать, каким образом функция может возвратить нецелое значение, рассмотрим функцию atof(s), которая переводит строку s в соответствующее число с плавающей точкой двойной точности. Она имеет дело со знаком (которого может и не быть), с десятичной точкой, а также с целой и дробной частями, одна из которых может отсутствовать. Функция atof входит в стандартную библиотеку программ: ее описание содержится в заголовочном файле <stdlib.h>.

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

```
double atof (char *s)      //atof: преобразование строки s в double
```

Кроме того, важно, чтобы вызывающая программа знала, что *atof* возвращает нецелое значение. Один из способов обеспечить это - явно описать *atof* в вызывающей программе.

В объявлении

```
double sum, atof (char *);
```

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

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

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

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

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

```
double atof();
```

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

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

```
int atoi (char *s)    // atoi: преобразование строки s в int с помощью atof

{
   double atof (char *s);
   return (int) atof (s);
}
```

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

```
return выражение;
```

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

4.7. Внешние переменные

Программа на Си обычно оперирует с множеством внешних объектов: переменных и функций. Прилагательное "внешний" (external) противоположно прилагательному "внутренний", которое относится к аргументам и переменным, определяемым внутри функций. Внешние переменные определяются вне

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

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

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

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

Переменная считается внешней, если она определена вне функции.

Если теперь программу представить как текст, расположенный в одном исходном файле, она будет иметь следующий вид:

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


объявления функций для main (прототипы)


main() {...}
внешние переменные для f1 и f2

void f1 (параметры) {...}
double f2 (параметры) {...}

int f3(параметры) {...}
```

4.8. Области видимости

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

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

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

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

• Как инициализировать внешние переменные?

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

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

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

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

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

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

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

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

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

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

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

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

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

В файле 1:

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

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

В файле2:

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

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

4.9. Заголовочные файлы

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

Существует еще один момент, о котором следует предупредить, - определения и объявления совместно используются несколькими файлами. Мы бы хотели, насколько это возможно, централизовать эти объявления и определения так, чтобы для них существовала только одна копия. Тогда программу в процессе ее развития будет легче и исправлять, и поддерживать в нужном состоянии. Для этого общую информацию расположим в заголовочном файле *calc.h*, который будем по мере необходимости включать в

другие файлы. (Строка #include описывается ниже) В результате получим программу, файловая структура которой показана ниже:

main.c:

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

calc.h:

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

getop.c:

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

getch.c:

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

stack.c:

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

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

4.10. Статические переменные

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

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

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

int getch(void) {...}

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

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

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

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

4.11. Регистровые переменные

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

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

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

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

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

4.12. Блочная структура

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

```
if (n > 0) {
   int i; /* описание новой переменной i */
```

```
    for (i = 0; i < n; i++){ тело цикла}
}
```

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

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

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

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

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

4.13. Инициализация

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

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

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

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

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

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

а не так:

```
int low, high, mid;

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

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

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

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

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

Если количество инициализаторов меньше числа, указанного в определении длины массива, то для внешних, статических и автоматических переменных оставшиеся элементы будут нулевыми. Задание

слишком большого числа инициализаторов считается ошибкой. В языке нет возможности ни задавать повторения инициализатора, ни инициализировать средние элементы массива без задания всех предшествующих значений. Инициализация символьных массивов - особый случай: вместо конструкции с фигурными скобками и запятыми можно использовать строку символов. Например, возможна такая запись:

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

представляющая собой более короткий эквивалент записи

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

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

4.14. Рекурсия

В Си допускается рекурсивное обращение к функциям, т. е. функция может обращаться сама к себе, прямо или косвенно.

```
    n = -n;
}
```

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

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

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

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

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

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

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

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

4.15. Препроцессор языка Си

Некоторые возможности языка Си обеспечиваются препроцессором, который работает на первом шаге компиляции. Наиболее часто используются две возможности: **#include**, вставляющая содержимое

некоторого файла во время компиляции, и **#define**, заменяющая одни текстовые последовательности на другие. Здесь обсудим условную компиляцию и макроподстановки с аргументами.

4.15.1. Включение файла

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

#include "*имя-файла*"

или

#include <*имя-файла*>

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

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

Средство *#include* - хороший способ собрать вместе объявления большой программы. Он гарантирует, что все исходные файлы будут пользоваться одними и теми же определениями и объявлениями переменных, благодаря чему предотвращаются особенно неприятные ошибки. Естественно, при внесении изменений во включаемый файл все зависимые от него файлы должны перекомпилироваться.

4.15.2. Макроподстановка

Определение макроподстановки имеет вид:

#define имя замещающий-текст

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

Любое имя можно определить с произвольным замещающим текстом. Например:

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

определяет новое слово *forever* для бесконечного цикла.

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

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

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

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

будет заменена на строку

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

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

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

max(i++, j++) /* НЕВЕРНО */

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

#define square(x) x*x /* НЕВЕРНО */

вызвать square (z+1).

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

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

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

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

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

Обращение к

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

развернется в

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

а в результате конкатенации двух соседних строк получим

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

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

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

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

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

Правила вложенных использований оператора ## не определены.

4.15.3. Условная компиляция

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

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

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

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

/* здесь содержимое hdr.h */

#endif
```

При первом включении файла *hdr.h* будет определено имя *HDR*, а при последующих включениях препроцессор обнаружит, что имя *HDR* уже определено, и перескочит сразу на *#endif*. Этот прием может оказаться полезным, когда нужно избежать многократного включения одного и того же файла. Если им пользоваться систематически, то в результате каждый заголовочный файл будет сам включать заголовочные файлы, от которых он зависит, освободив от этого занятия пользователя.

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

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

```
#endif
#include HDR
```

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

```
#ifndef HDR
#define HDR

/* здесь содержимое hdr.h */

#endif
```

4.16. Указатели и массивы

Указатель - это переменная, содержащая адрес переменной. Указатели широко применяются в Си - отчасти потому, что в некоторых случаях без них просто не обойтись, а отчасти потому, что программы с ними обычно короче и эффективнее. Указатели и **массивы** тесно связаны друг с другом: в данной главе мы рассмотрим эту зависимость и покажем, как ею пользоваться. Наряду с *goto* указатели когда-то были объявлены лучшим средством для написания малопонятных программ. Так оно и есть, если ими пользоваться бездумно. Ведь очень легко получить указатель, указывающий на что-нибудь совсем нежелательное. При соблюдении же определенной дисциплины с помощью указателей можно достичь ясности и простоты. Мы попытаемся убедить вас в этом.

Изменения, внесенные стандартом ANSI, связаны в основном с формулированием точных правил, как работать с указателями. Стандарт узаконил накопленный положительный опыт программистов и удачные нововведения разработчиков компиляторов. Кроме того, взамен *char** в качестве типа обобщенного указателя предлагается тип *void** (указатель на void).

Для чего нужны указатели? Вот наиболее частые примеры их использования:

- Доступ к элементам массива
- Передача аргументов в функцию, от которой требуется изменить эти аргументы
- Передача в функции массивов и строковых переменных
- Выделение памяти
- Создание сложных структур, таких, как связный список или бинарное дерево.

Идея указателей несложна. Каждый байт в памяти машины имеет свой уникальный адрес. Адреса начинаются с 0, а затем монотонно возрастают. Если у нас есть 1 Мб ОП, то максимальным адресом будет число 1 048 575.

Загружаясь в память, наша программа занимает некоторое количество этих адресов. Это означает, что каждая переменная и каждая функция нашей программы начинается с некоторого конкретного адреса.

4.16.1.Операция получения адреса &

Мы можем получить адрес переменной, используя операцию получения адреса &.

```
Int x1=11, x2=22, x3=33;
```

&x1 - возвращает адрес переменной x1;

&x2 - возвращает адрес переменной x2;

&x3 - возвращает адрес переменной x3;

Реальные адреса, занятые в программе, зависят от многих факторов, таких, как компьютер, на котором запущена программа, размер ОП, наличие другой программы в памяти и т.д. По этой причине от запуска к запуску программа будет возвращать в общем случае разные адреса. Пусть мы получили следующие значения

Ox8f4ffff4 - адрес переменной x1;

Ox8f4ffff2 - адрес переменной x2;

Ox8f4ffff0 - адрес переменной x3;

Важно понимать, что адрес переменной - это не то же самое, что значение переменной. Содержимое трех переменных - это числа 11, 22 и 33.

Заметим, что адреса отличаются друг от друга двумя байтами. Это произошло потому, что переменная типа int занимает в памяти два байта. Если бы мы использовали переменные типа char, то значения адресов отличались бы на единицу.

4.16.2. Переменные указатели

Адресное пространство ограничено. Нам необходимы переменные, хранящие значение адреса. Нам знакомы переменные, хранящие знаки, целые или вещественные числа и т.д. Адреса хранятся точно так же. Переменная, содержащая в себе значение адреса, называется переменной-указателем или просто указателем.

Какого же типа может быть переменная-указатель? Она не того же типа, что и переменная, адрес которой мы храним: указатель на int не имеет типа int.

Int *ptr; - определение переменной-указателя на целое. * - означает *указатель на.* Т.е. переменная ptr может содержать в себе адрес переменной типа int.

Компилятору необходимы сведения о том, какого именно типа переменная, на которую указывает указатель. Поэтому не существует обобщенный тип pointer такой, чтобы мы могли записать, например

pointer ptr;

4.16.3. Указатели должны иметь значение

Пусть у нас есть адрес Ox8f4ffff4, мы можем назвать его значением указателя. Указатель ptr называется переменной-указателем. Как переменная int xl может принимать значение, равное 11, так переменная-указатель может принимать значение, равное Ox8f4ffff4.

Пусть мы определили некоторую переменную для хранения некоторого значения, но не проинициализировали ее. Она будет содержать некоторое случайное число. В случае с переменной-указателем это случайное число является некоторым адресом в памяти. Этот адрес может оказаться чем угодно: от кода нашей программы до кода ОС. Неинициализированный указатель может привести к краху ОС, его очень тяжело выявить при отладке программы, т.к. компилятор не выдает предупреждения о подобных ошибках. Поэтому всегда важно убедиться, что каждому указателю перед его использованием было присвоено значение, т.е. его необходимо проинициализировать определенным адресом, например ptr =&xl.

4.16.4. Доступ к переменной по указателю

Существует запись, позволяющая получить доступ к значению переменной по ее адресу

```
int xl=ll,x2=22;
int *ptr;

ptr = &xl;
printf("%d",*ptr); //печать содержимого через указатель
ptr = &x2;
printf("%d",*ptr);
```

Выражение *ptr дважды встречается в нашей программе, позволяет нам получить значения переменных xl и x2.

*, стоящая перед именем переменной, как в выражении *ptr, называется *операцией разыменования.* Эта запись означает: взять значение переменной, на которую указывает указатель. Таким образом, выражение *ptr представляет собой значение переменной, на которую указывает указатель ptr.

*, используемая в операции разыменования,- это не то же самое, что звездочка, используемая при объявлении указателя. Операция разыменования предшествует имени переменной и означает значение, находящееся в переменной, на которую указывает указатель. * в объявлении указателя означает *указатель на.*

Доступ к значению переменной, хранящейся по адресу, с использованием операции разыменования называется непрямым доступом или разыменованием указателя.

4.16.5 Указатель на void

Отметим одну особенность указателей. Адрес, который мы помещаем в указатель, должен быть одного с ним типа. Мы не можем присвоить указателю на int адрес переменной типа float.

```
float fx=98.6;
int ptr = &fx; //так нельзя, типы int * и float * несовместны
```

Однако есть одно исключение. Существует тип указателя, который может указывать на любой тип данных. Он называется указателем на тип void и определяется следующим образом:

```
void *ptr.
```

Такие указатели предназначены для использования в определенных случаях, например, передача указателей в функции, работающие независимо от типа данных, на которые указывает указатель.

```
int ix;
float fx;
int *iptr;
float *fptr;
void *vptr;
iptr =&ix;
fptr = &fx;
vptr = &ix;
vptr = &fx;
```

Следующие строки недопустимы

```
iptr - &fx;
fptr = &ix;
```

4.16.6. Указатели-константы и указатели переменные

Можно ли записать *(array++), т.е использовать операцию увеличения вместо прибавления шага j к имени array? Нет, так писать нельзя, поскольку нельзя изменять константы. Выражение array является адресом в памяти, где размещен наш массив, поэтому array - это указатель
константы. Мы не можем сказать array++, так же как не можем сказать 7++. Мы не можем увеличивать адрес, но имеем возможность увеличить указатель, который содержит этот адрес.

```
int main()
    {
        int array[5]={3 1,54,77,52,93};
        int j;
        int *prt;
        prt = array;

        for(j=0; j<5; j++) printf("%6d",*(prt++);
        return 0;
    }
```

Здесь мы определили указатель на int и затем присвоили ему значение адреса массива. После этого мы можем получить доступ к элементам массива, используя операцию разыменования *(prt++). Переменная

prt имеет тот же адрес, что и array, поэтому доступ к первому элементу осуществляется как и раньше. Но prt - это уже переменная-указатель, то мы ее можем увеличивать. После этого она уже будет указывать на первый элемент массива array.

4.16.7 Передача простой переменной в функцию

Рассмотрим функцию void summ(int a, int b, int *us). 3-ий параметр объявлен в этой функции как указатель на int. Когда головная программа вызывает функцию summ, то она передает в качестве 3 аргумента адрес переменной summ(a, b,&s). Это не сама переменная, а ее адрес. Т.к. функция summ получает адрес переменной s, то она может применить к ней операцию разыменования *us = a+b. Т.к. us содержит адрес переменной s, то все действия, которые мы совершим с *us, мы в действительности совершим с s.

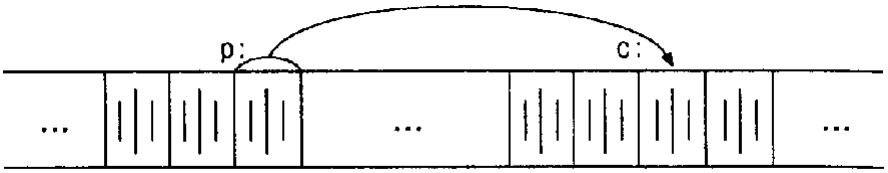
4.16.8. Передача массивов

Пусть мы хотим передать в функцию массив int A[10]. Прототип такой функции может выглядеть следующим образом void func(int[]). В этом случае мы обеспечиваем передачу массива по значению. Если мы хотим использовать аппарат указателей, то передача аргумента будет выглядеть как void func(int *).

Т.к. имя массива является его адресом, то нет нужды использовать при вызове функции операцию взятия адреса, т.е можно записать func(A).

4.16.9.Указатели и адреса

Начнем с того, что рассмотрим упрощенную схему организации памяти. Память типичной машины подставляет собой массив последовательно пронумерованных или проадресованных ячеек, с которыми можно работать по отдельности или связными кусками. Применительно к любой машине верны следующие утверждения: один байт может хранить значение типа *char*, двухбайтовые ячейки могут рассматриваться как целое типа *short*, а четырехбайтовые - как целые типа *long*. Указатель - это группа ячеек (как правило, две или четыре), в которых может храниться адрес. Так, если с имеет тип *char*, а *p* - указатель на *c*, то ситуация выглядит следующим образом:



Унарный оператор **&** выдает адрес объекта, так что инструкция

p = &c;

присваивает переменной *p* адрес ячейки *c* (говорят, что *p* указывает на *c*). Оператор **&** применяется только к объектам, расположенным в памяти: к переменным и элементам массивов. Его операндом не может быть ни выражение, ни константа, ни регистровая переменная.

Унарный оператор * есть оператор *косвенного доступа*. Примененный к указателю он выдает объект, на который данный указатель указывает. Предположим, что *x* и *y* имеют тип *int*, а *ip* – укаэатель на *int*. Следующие несколько строк придуманы специально для того, чтобы показать, каким образом объявляются указатели и как используются операторы **&** и *.

```
int x = 1, y = 2, z[10];
int *ip;     ip - указатель на int

ip = &x;    теперь ip указывает на x
y = *ip;     y теперь равен 1
*ip = 0;     x теперь равен 0
ip = &z[0];  ip теперь указывает на z[0]
```

Объявления *x*, *y* и *z* нам уже знакомы. Объявление указателя *ip*

int *ip;

мы стремились сделать мнемоничным - оно гласит: "выражение **ip* имеет тип *int*". Синтаксис объявления переменной "подстраивается" под синтаксис выражений, в которых эта переменная может встретиться. Указанный принцип применим и в объявлениях функций. Например, запись

double *dp, atof (char *);

означает, что выражения **dp* и *atof(s)* имеют тип *double*, а аргумент функции *atof* есть указатель на *char*.

Вы, наверное, заметили, что указателю разрешено указывать только на объекты определенного типа. (Существует одно исключение: "указатель на *void*" может указывать на объекты любого типа, но к такому указателю нельзя применять оператор косвенного доступа)

Если *ip* указывает на *x* целочисленного типа, то **ip* можно использовать в любом месте, где допустимо применение *x*; например,

```
*ip = *ip + 10;
```

увеличивает **ip* на 10.

Унарные операторы **** и **&** имеют более высокий приоритет, чем арифметические операторы, так что присваивание

```
y = *ip + 1;
```

берет то, на что указывает *ip*, и добавляет к нему 1, а результат присваивает переменной *y*. Аналогично

```
*ip += 1;
```

увеличивает на единицу то, на что указывает *ip*; те же действия выполняют

```
++*ip;
```

и

```
(*ip)++;
```

В последней записи скобки необходимы, поскольку если их не будет, увеличится значение самого указателя, а не то, на что он указывает. Это обусловлено тем, что унарные операторы * и ++ имеют одинаковый приоритет и порядок выполнения - справа налево.

И наконец, так как указатели сами являются переменными, в тексте они могут встречаться и без оператора косвенного доступа. Например, если *iq* есть другой указатель на *int*, то

```
iq = ip;
```

копирует содержимое *ip* в *iq*, чтобы *ip* и *iq* указывали на один и тот же объект.

4.16.10. Указатели и аргументы функций

Поскольку в Си функции в качестве своих аргументов получают значения параметров, нет прямой возможности, находясь в вызванной функции, изменить переменную вызывающей функции. В программе сортировки нам понадобилась функция *swap*, меняющая местами два неупорядоченных элемента. Однако недостаточно написать

```
swap(a, b);
```

где функция *swap* определена следующим образом:

```
void swap(int x, int y) /* НЕВЕРНО */
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

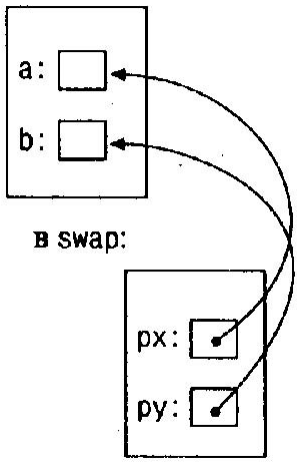
Поскольку swap получает лишь копии переменных a и b, она не может повлиять на переменные a и b той программы, которая к ней обратилась. Чтобы получить желаемый эффект, вызывающей программе надо передать указатели на те значения, которые должны быть изменены:

```
swap(&a, &b);
```

Так как оператор **&** получает адрес переменной, *&a* есть указатель на *a*. В самой же функции *swap* параметры должны быть объявлены как указатели, при этом доступ к значениям параметров будет осуществляться косвенно.

```
void swap(int *px, int *py) /* перестановка *px и *py */
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```

Графически это выглядит следующим образом: в вызывающей программе:



Аргументы-указатели позволяют функции осуществлять доступ к объектам вызвавшей ее программы и дают возможность изменить эти объекты.

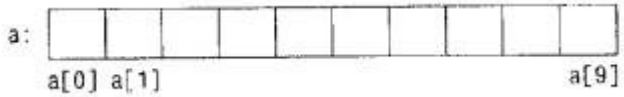
4.16.11. Указатели и массивы

В Си существует связь между указателями и массивами, и связь эта настолько тесная, что эти средства лучше рассматривать вместе. Любой доступ к элементу массива, осуществляемый операцией индексирования, может быть выполнен с помощью указателя. Вариант с указателями в общем случае работает быстрее, но разобраться в нем, особенно непосвященному, довольно трудно.

Объявление

int a[10];

Определяет массив *a* размера 10, т. е. блок из 10 последовательных объектов с именами a[0], a[1], ..., a[9].



Запись *a[i]* отсылает нас к *i*-му элементу массива. Если *pa* есть указатель на *int*, т. е. объявлен как

int *pa;

то в результате присваивания

pa = &a[0];

pa будет указывать на нулевой элемент *a*, иначе говоря, *pa* будет содержать адрес элемента *a[0]*.

Теперь присваивание

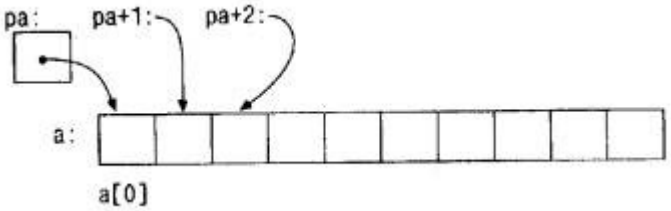
x = *pa;

будет копировать содержимое *a[0]* в *x*.

Если *pa* указывает на некоторый элемент массива, то *pa+1* по определению указывает на следующий элемент, *pa+i* - на *i*-й элемент после *pa*, а *pa-i* - на *i*-й элемент перед *pa*. Таким образом, если *pa* указывает на *a[0]*, то

*(pa+1)

есть содержимое *a[1]*, *a+i* - адрес *a[i]*, а **(pa+i)* - содержимое *a[i]*.



Сделанные замечания верны безотносительно к типу и размеру элементов массива *a*. Смысл слов "добавить 1 к указателю", как и смысл любой арифметики с указателями, состоит в том, чтобы *pa+1* указывал на следующий объект, а *pa+i* - на *i*-й после *pa*.

Между индексированием и арифметикой с указателями существует очень тесная связь. По определению значение переменной или выражения типа массив есть адрес нулевого элемента массива. После присваивания

pa = &a[0];

pa и *a* имеют одно и то же значение. Поскольку имя массива является синонимом расположения его начального элемента, присваивание *pa=&a[0]* можно также записать в следующем виде:

pa = a;

Еще более удивительно (по крайней мере на первый взгляд) то, что *a[i]* можно записать как **(a+i)*. Вычисляя *a[i]*, Си сразу преобразует его в **(a+i)*; указанные две формы записи эквивалентны. Из этого следует, что полученные в результате применения оператора *&* записи *&a[i]* и *a+i* также будут эквивалентными, т. е. и в том и в другом случае это адрес *i*-го элемента после *a*. С другой стороны, если *pa* - указатель, то его можно использовать с индексом, т. е. запись *pa[i]* эквивалентна записи **(pa+i)*. Короче говоря, элемент массива можно изображать как в виде указателя со смещением, так и в виде имени массива с индексом.

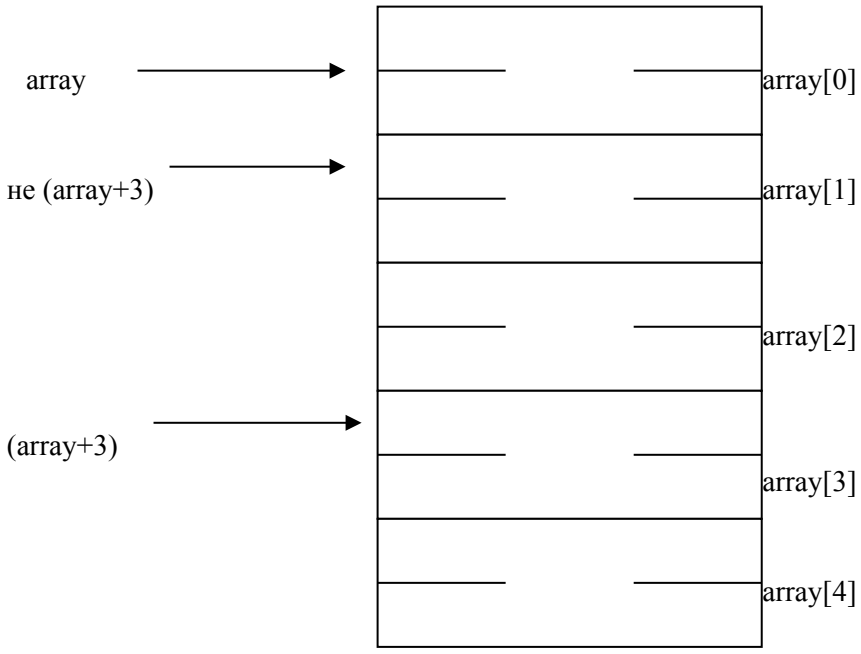
Между именем массива и указателем, выступающим в роли имени массива, существует одно различие. *Указатель - это переменная*, поэтому можно написать *pa=a* или *pa++*. Но *имя массива не является переменной*, и записи вроде *a=pa* или *a++* не допускаются.

Указатели и массивы очень похожи. Оказывается, что доступ к элементам массива можно получить как используя операции с массивами, так и используя указатели. Рассмотрим два способа доступа к элементам массива

```
int main()
{
int array[5]={31,54,77,52,93};
    int j;

    for(j=0;j<5;j++){printf("%6d",arraj[j]); printf("%6d",*(arra,+j);}
    return 0;
}
```

Заметим, что результат действия выражения *(array+j) тот же, что и выражения arraj[j]. Вспомним, что имя массива является его адресом. Таким образом, выражение array+j – это адрес чего-то в массиве. Пусть j=3, тогда можно ожидать, что array+j будет означать 3 байта массива array. Поскольку array - это массив элементов типа int, и три байта в этом массиве - середина второго элемента, что не очень полезно для нас. Мы хотим получить четвертый элемент массива, а не его четвертый байт.



Компилятору С достаточно получить размер данных в счетчике для выполнения вычислений с адресами данных. Ему известно, что array - массив типа int. Поэтому, видя выражение array+3, компилятор интерпретирует его как адрес четвертого элемента массива, а е четвертого байта.

Но нам необходимо значение четвертого элемента, а не его адрес. Для его получения мы используем операцию разыменования(*). Поэтому результатом выполнения выражения *(array+3) будет значение четвертого элемента массива, то есть 52.

Теперь становится понятно, почему при объявлении указателя мы должны указать тип переменной, на которую он будет указывать. Компилятору необходимо знать, на переменные какого типа указатель указывает, чтобы осуществлять правильный доступ к элементам массива.Он умножает значение индекса на 2, в случае типа int, или на 4 - в случае типа float и т.д.

Если имя массива передается функции, то последняя получает в качестве аргумента адрес его начального элемента. Внутри вызываемой функции этот аргумент является локальной переменной, содержащей адрес.

Формальные параметры

char s[];

и

char *s;

в определении функции эквивалентны. Мы отдаем предпочтение последнему варианту, поскольку он более явно сообщает, что s есть указатель. Если функции в качестве аргумента передается имя массива, то она может рассматривать его так, как ей удобно - либо как имя массива, либо как указатель, и поступать с ним соответственно. Она может даже использовать оба вида записи, если это покажется уместным и понятным.

Функции можно передать часть массива, для этого аргумент должен указывать на начало подмассива. Например, если *a* - массив, то в записях

f(&a[2])

или

f(a+2)

функции *f* передается адрес подмассива, начинающегося с элемента *a[2]*. Внутри функции *f* описание параметров может выглядеть как

f(int arr[]) {...}

или

f(int *arr) {...}

Следовательно, для *f* тот факт, что параметр указывает на часть массива, а не на весь массив, не имеет значения.

Если есть уверенность, что элементы массива существуют, то возможно индексирование и в "обратную" сторону по отношению к нулевому элементу; выражения *p[-1]*, *p[-2]* и т.д. не противоречат синтаксису языка и обращаются к элементам, стоящим непосредственно перед *p[0]*. Разумеется, нельзя "выходить" за границы массива и тем самым обращаться к несуществующим объектам.

4.16.12. Адресная арифметика

Если *p* есть указатель на некоторый элемент массива, то *p++* увеличивает *p* так, чтобы он указывал на следующий элемент, а *p+=i* увеличивает его, чтобы он указывал на *i*-й элемент после того, на который указывал ранее. Эти и подобные конструкции - самые простые примеры арифметики над указателями, называемой также адресной арифметикой.

Си последователен и единообразен в своем подходе к адресной арифметике. Это соединение в одном языке указателей, массивов и адресной арифметики - одна из сильных его сторон.

В общем случае указатель, как и любую другую переменную, можно инициализировать, но только такими осмысленными для него значениями, как нуль или выражение, приводящее к адресу ранее определенных данных соответствующего типа.

#define ALLOCSIZE 10000 /* размер доступного пространства */

static char allocbuf[ALLOCSIZE]; /* память для alloc */

static char *allocp = allocbuf; /* указатель на своб. место */

Объявление

static char *allocp = allocbuf;

определяет *allocp* как указатель на *char* и инициализирует его адресом массива *allocbuf*, поскольку перед началом работы программы массив *allocbuf* пуст. Указанное объявление могло бы иметь и такой вид:

static char *allocp = &allocbuf[0];

поскольку имя массива и есть адрес его нулевого элемента.

Си гарантирует, что нуль никогда не будет правильным адресом для данных, поэтому мы будем использовать его в качестве признака аварийного события, в нашем случае нехватки памяти.

Указатели и целые не являются взаимозаменяемыми объектами. Константа нуль - единственное исключение из этого правила: ее можно присвоить указателю, и указатель можно сравнить с нулевой константой. Чтобы показать, что нуль - это специальное значение для указателя, вместо цифры нуль, как правило, записывают **NULL** - константу, определенную в файле <stdio.h>.

Несколько важных свойств арифметики с указателями.

Во- первых, при соблюдении некоторых правил указатели можно сравнивать.

Если *p* и *q* указывают на элементы одного массива, то к ним можно применять операторы отношения ==, **!=**, **<**, **>=** и т. д. Например, отношение вида

p < q

истинно, если *p* указывает на более ранний элемент массива, чем *q*. Любой указатель всегда можно сравнить на равенство и неравенство с нулем. А вот для указателей, не указывающих на элементы одного массива, результат арифметических операций или сравнений не определен. (Существует одно исключение: в арифметике с указателями можно использовать адрес несуществующего "следующего за массивом" элемента, т. е. адрес того "элемента", который станет последним, если в массив добавить еще один элемент.)

Во-вторых, указатели и целые можно складывать и вычитать. Конструкция

p + n

означает адрес объекта, занимающего *n*-е место после объекта, на который указывает *p*. Это справедливо безотносительно к типу объекта, на который указывает *p*; *n* автоматически домножается на коэффициент, соответствующий размеру объекта. Информация о размере неявно присутствует в объявлении *p*. Если, к примеру, *int* занимает четыре байта, то коэффициент умножения будет равен четырем.

Допускается также вычитание указателей. Например, если *p* и *q* указывают на элементы одного массива и *p<q*, то *q-p+1* есть число элементов от *p* до *q* включительно

Арифметика с указателями учитывает тип: если она имеет дело со значениями *float*, занимающими больше памяти, чем *char*, и *p* - указатель на *float*, то *p++* продвинет *p* на следующее значение *float*. Это значит, что другую версию *alloc*, которая имеет дело с элементами типа *float*, а не *char*, можно получить простой заменой в *alloc* и *afree* всех *char* на *float*. Все операции с указателями будут автоматически откорректированы в соответствии с размером объектов, на которые указывают указатели.

Можно производить следующие операции с указателями:

- присваивание значения указателя другому указателю того же типа,
- сложение и вычитание указателя и целого,
- вычитание и сравнение двух указателей, указывающих на элементы одного и того же массива,
- присваивание указателю нуля и сравнение указателя с нулем.

Других операций с указателями производить не допускается.

Нельзя складывать два указателя, перемножать их, делить, сдвигать, выделять разряды; указатель нельзя складывать со значением типа *float* или *double*; указателю одного типа нельзя даже присвоить указатель другого типа, не выполнив предварительно операции приведения (исключение составляют лишь указатели типа void*).

4.16.13. Символьные указатели функции

Строковая константа, написанная в виде

"Я строка"

есть массив символов. Во внутреннем представлении этот массив заканчивается нулевым символом '\0', по которому программа может найти конец строки. Число занятых ячеек памяти на одну больше, чем количество символов, помещенных между двойными кавычками.

Чаще всего строковые константы используются в качестве аргументов функций, как, например, в

printf("здравствуй, мир\n");

Когда такая символьная строка появляется в программе, доступ к ней осуществляется через символьный указатель; *printf* получает указатель на начало массива символов. Точнее, доступ к строковой константе осуществляется через указатель на ее первый элемент.

Строковые константы нужны не только в качестве аргументов функций. Если, например, переменную *pmessage* объявить как

char *pmessage;

то присваивание

pmessage = "now is the time";

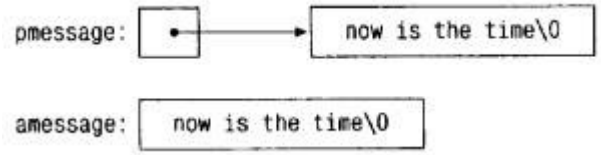
поместит в нее указатель на символьный массив, при этом сама строка не копируется, копируется лишь указатель на нее. Операции для работы со строкой как с единым целым в Си не предусмотрены.

Существует важное различие между следующими определениями:

char amessage[] = "now is the time"; /* массив */

char *pmessage = "now is the time"; /* указатель */

amessage - это массив, имеющий такой объем, что в нем как раз помещается указанная последовательность символов и '\0'. Отдельные символы внутри массива могут изменяться, но *amessage* всегда указывает на одно и то же место памяти. В противоположность ему *pmessage* есть указатель, инициализированный так, чтобы указывать на строковую константу. А значение указателя можно изменить, и тогда последний будет указывать на что-либо другое. Кроме того, результат будет неопределен, если вы попытаетесь изменить содержимое константы.



Объявления стандартных функций, работающих со строками, содержатся в заголовочном файле **<string.h>**.

4.16.14. Многомерные массивы

В Си имеется возможность задавать прямоугольные многомерные массивы, правда, на практике по сравнению с массивами указателей они используются значительно реже. Рассмотрим некоторые их свойства.

Особенность двумерного массива в Си заключается лишь в форме записи, в остальном его можно трактовать почти так же, как в других языках. Элементы запоминаются строками, следовательно, при переборе их в том порядке, как они расположены в памяти, чаще будет изменяться самый правый индекс.

Массив инициализируется списком начальных значений, заключенным в фигурные скобки; каждая строка двумерного массива инициализируется соответствующим подсписком.

Например,

```
Int A[2][3]={
              {0,0,0};
              {1,1,1};
            };
```

Если двумерный массив передается функции в качестве аргумента, то объявление соответствующего ему параметра должно содержать количество столбцов; количество строк в данном случае несущественно, поскольку, как и прежде, функции будет передан указатель на массив строк, каждая из которых есть массив из 13 значений типа *int*. В нашем частном случае мы имеем указатель на объекты, являющиеся массивами из 13 значений типа *int*. Таким образом, если массив *daytab* передается некоторой функции *f*, то эту функцию можно было бы определить следующим образом:

f(int daytab[2][13]) {...}

Вместо этого можно записать

f(int daytab[][13]) {...}

поскольку число строк здесь не имеет значения, или

f(int (*daytab)[13]) {...}

Последняя запись объявляет, что параметр есть указатель на массив из 13 значений типа int. Скобки здесь необходимы, так как квадратные скобки [] имеют более высокий приоритет, чем *. Без скобок объявление

int *daytab[13]

определяет массив из 13 указателей на char. В более общем случае только первое измерение (соответствующее первому индексу) можно не задавать, все другие специфицировать необходимо.

4.16.15. Указатели против многомерных массивов

Начинающие программировать на Си иногда не понимают, в чем разница между двумерным массивом и массивом указателей. Для двух следующих определений:

int a[10][20];

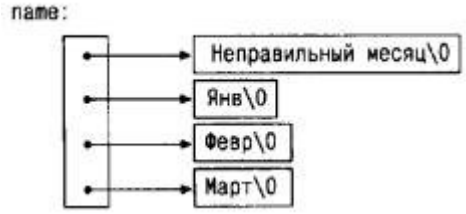
int *b[10];

записи *a[3][4]* и *b[3][4]* будут синтаксически правильным обращением к некоторому значению типа *int*. Однако только *a* является истинно двумерным массивом: для двухсот элементов типа *int* будет выделена память, а вычисление смещения элемента *a[строка][столбец]* от начала массива будет вестись

по формуле *20 * строка + столбец*, учитывающей его прямоугольную природу. Для *b* же определено только 10 указателей, причем без инициализации. Инициализация должна задаваться явно -либо статически, либо в программе. Предположим, что каждый элемент *b* указывает на двадцатиэлементный массив, в результате где-то будут выделены пространство, в котором разместятся 200 значений типа *int*, и еще 10 ячеек для указателей. Важное преимущество массива указателей в том, что строки такого массива могут иметь разные длины. Таким образом, каждый элемент массива *b* не обязательно указывает на двадцатиэлементный вектор; один может указывать на два элемента, другой - на пятьдесят, а некоторые и вовсе могут ни на что не указывать.

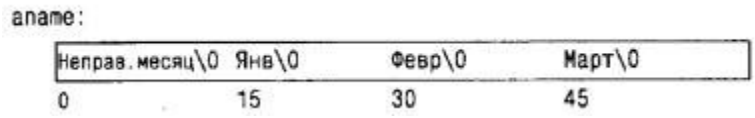
Наши рассуждения здесь касались целых значений, однако чаще массивы указателей используются для работы со строками символов, различающимися по длине, как это было в функции *month_name*. Сравните определение массива указателей и соответствующий ему рисунок:

char *name[] = {"Неправильный месяц", "Янв", "Февр", "Март"};



с объявлением и рисунком для двумерного массива:

char aname[][15] = {"Неправ. месяц", "Янв", "Февр", "Март"};



4.16.16. Аргументы командной строки

В операционной среде, обеспечивающей поддержку Си, имеется возможность передать аргументы или параметры запускаемой программе с помощью командной строки. В момент вызова *main* получает два аргумента. В первом, обычно называемом *argc* (сокращение от *argument count*), стоит количество аргументов, задаваемых в командной строке. Второй, *argv* (от *argument vector*), является указателем на массив символьных строк, содержащих сами аргументы. Для работы с этими строками обычно используются указатели нескольких уровней.

```
int main(int argc, char *argv[])
```

Аргумент *argv* - указатель на начало массива строк аргументов.По соглашению *argv[0]* есть имя вызываемой программы, так что значение *argc* никогда не бывает меньше 1. Если *argc* равен 1, то в командной строке после имени программы никаких аргументов нет. Кроме того, стандарт требует, чтобы *argv[argc]* всегда был пустым указателем.

Так как *argv* - это указатель на массив указателей, мы можем работать с ним как с указателем, а не как с индексируемым массивом.

4.16.17. Указатели на функции

В Си сама функция не является переменной, но можно определить указатель на функцию и работать с ним, как с обычной переменной: присваивать, размещать в массиве, передавать в качестве параметра функции, возвращать как результат из функции и т. д. Для иллюстрации этих возможностей воспользуемся программой сортировки, которая уже встречалась в настоящей главе. Изменим ее так, чтобы при задании необязательного аргумента -n вводимые строки упорядочивались по их числовому значению, а не в лексикографическом порядке.

Сортировка, как правило, распадается на три части: на сравнение, определяющее упорядоченность пары объектов; перестановку, меняющую местами пару объектов, и сортирующий алгоритм, который осуществляет сравнения и перестановки до тех пор, пока все объекты не будут упорядочены. Алгоритм сортировки не зависит от операций сравнения и перестановки, так что передавая ему в качестве параметров различные функции сравнения и перестановки, его можно настроить на различные критерии сортировки.

Лексикографическое сравнение двух строк выполняется функцией *strcmp* (мы уже использовали эту функцию в ранее рассмотренной программе сортировки); нам также потребуется программа *numcmp*, сравнивающая две строки как числовые значения и возвращающая результат сравнения в том же виде, в каком его выдает *strcmp*. Эти функции объявляются перед *main*, а указатель на одну из них передается

функции *qsort*. Чтобы сосредоточиться на главном, мы упростили себе задачу, отказавшись от анализа возможных ошибок при задании аргументов.

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

#define MAXLINES 5000 /* максимальное число строк */
char *lineptr[MAXLINES]; /* указатели на строки текста */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void qsort(void *lineptr[], int left, int right,
   int (*comp)(void *, void *));
int numcmp(char *, char *);

/* сортировка строк */
main(int argc, char *argv[])
{
   int nlines;        /* количество прочитанных строк */
   int numeric = 0;   /* 1, если сорт. по числ. знач. */
   if (argc > 1 && strcmp(argv[1], "-n") == 0)
      numeric = 1;
   if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
      qsort((void **) lineptr, 0, nlines-1,
        (int (*)(void*,void*))(numeric ? numcmp : strcmp));
      writelines(lineptr, nlines);
      return 0;
   } else {
      printf("Введено слишком много строк\n");
      return 1;
   }
}
```

В обращениях к функциям *qsort*, *strcmp* и *numcmp* их имена трактуются как адреса этих функций, поэтому оператор & перед ними не нужен, как он не был нужен и перед именем массива.

Мы написали *qsort* так, чтобы она могла обрабатывать данные любого типа, а не только строки символов. Как видно из прототипа, функция *qsort* в качестве своих аргументов ожидает массив указателей, два целых значения и функцию с двумя аргументами-указателями. В качестве аргументов-указателей заданы указатели обобщенного типа *void* *. Любой указатель можно привести к типу *void* * и обратно без потери информации, поэтому мы можем обратиться к *qsort*, предварительно преобразовав аргументы в *void* *. Внутри функции сравнения ее аргументы будут приведены к нужному ей типу. На самом деле эти преобразования никакого влияния на представления аргументов не оказывают, они лишь обеспечивают согласованность типов для компилятора.

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

   if (left >= right) /* ничего не делается, если */
      return;        /* в массиве менее двух элементов */
   swap(v, left, (left + right)/2);
   last = left;
   for (i = left+1; i <= right; i++)
      if ((*comp)(v[i], v[left]) < 0)
         swap(v, ++last, i);
   swap(v, left, last);
   qsort(v, left, last-1, comp);
   qsort(v, last+1, right, comp);
}
```

Повнимательней приглядимся к объявлениям. Четвертый параметр функции qsort:

int (*comp)(void *, void *)

сообщает, что *comp* - это указатель на функцию, которая имеет два аргумента- указателя и выдает результат типа *int*. Использование *comp* в строке

if ((*comp)(v[i], v[left]) < 0)

согласуется с объявлением "*comp* - это указатель на функцию", и, следовательно, **comp* - это функция, а

(*comp)(v[i], v[left])

- обращение к ней. Скобки здесь нужны, чтобы обеспечить правильную трактовку объявления; без них объявление

int *comp(void *, void *) /* НЕВЕРНО */

говорило бы, что *comp* - это функция, возвращающая указатель на *int*, а это совсем не то, что требуется.

Мы уже рассматривали функцию *strcmp*, сравнивающую две строки. Ниже приведена функция *numcmp*, которая сравнивает две строки, рассматривая их как числа; предварительно они переводятся в числовые значения функцией *atof*.

```
#include <stdlib.h>

/* numcmp: сравнивает s1 и s2 как числа */

int numcmp(char *s1, char *s2)
{
   double v1, v2;

   v1 = atof(s1);
   v2 = atof(s2);
   if (v1 < v2)
      return -1;
   else if (v1 > v2)
      return 1;
   else
      return 0;
}
```

Функция swap, меняющая местами два указателя, идентична той, что мы привели ранее в этой главе за исключением того, что объявления указателей заменены на void*.

```
void swap(void *v[], int i, int j)
{
   void *temp;
   temp = v[i];
   v[i] = v[j];
   v[j] = temp;
}
```

Программу сортировки можно дополнить и множеством других возможностей; реализовать некоторые из них предлагается в качестве упражнений.

4.17. Структуры

Структура - это одна или несколько переменных (возможно, различных типов), которые для удобства работы с ними сгруппированы под одним именем. (В некоторых языках, в частности в Паскале, структуры называются записями.) Структуры помогают в организации сложных данных (особенно в больших программах), поскольку позволяют группу связанных между собой переменных трактовать не как множество отдельных элементов, а как единое целое.

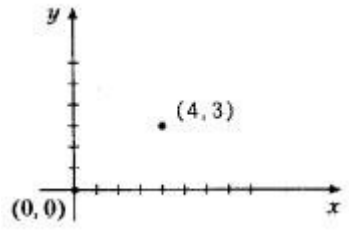
Традиционный пример структуры - строка платежной ведомости. Она содержит такие сведения о служащем, как его полное имя, адрес, номер карточки социального страхования, зарплата и т. д. Некоторые из этих характеристик сами могут быть структурами: например, полное имя состоит из нескольких компонент (фамилии, имени и отчества); аналогично адрес, и даже зарплата. Другой пример (более типичный для Си) - из области графики: точка есть пара координат, прямоугольник есть пара точек и т. д.

Главные изменения, внесенные стандартом ANSI в отношении структур, - это введение для них операции присваивания. Структуры могут копироваться, над ними могут выполняться операции присваивания, их можно передавать функциям в качестве аргументов, а функции могут возвращать их в качестве результатов. В большинстве компиляторов уже давно реализованы эти возможности, но теперь

они точно оговорены стандартом. Для автоматических структур и массивов теперь также допускается инициализация.

4.17.1. Основные сведения о структурах

Сконструируем несколько графических структур. В качестве основного объекта выступает точка с координатами *x* и *y* целого типа.



Указанные две компоненты можно поместить в структуру, объявленную, например, следующим образом:

```
struct point {
   int x;
   int y;
};
```

Объявление структуры начинается с ключевого слова **struct** и содержит список объявлений, заключенный в фигурные скобки. За словом **struct** может следовать имя, называемое *тегом структуры* (от английского слова tag — ярлык, этикетка.), *point* в нашем случае. Тег дает название структуре данного вида и далее может служить кратким обозначением той части объявления, которая заключена в фигурные скобки.

Перечисленные в структуре переменные называются *элементами* - *members*. Имена элементов и тегов без каких-либо коллизий могут совпадать с именами обычных переменных (т. е. не элементов), так как они всегда различимы по контексту. Более того, одни и те же имена элементов могут встречаться в разных структурах, хотя, если следовать хорошему стилю программирования, лучше одинаковые имена давать только близким по смыслу объектам.

Объявление структуры определяет тип. За правой фигурной скобкой, закрывающей список элементов, могут следовать переменные точно так же, как они могут быть указаны после названия любого базового типа. Таким образом, выражение

```
struct {...} x, y, z;
```

с точки зрения синтаксиса аналогично выражению

```
int x, y, z;
```

в том смысле, что и то и другое объявляет x, y и z переменными указанного типа; и то и другое приведет к выделению памяти соответствующего размера.

Объявление структуры, не содержащей списка переменных, не резервирует памяти; оно просто описывает шаблон, или образец структуры. Однако если структура имеет тег, то этим тегом далее можно пользоваться при определении структурных объектов. Например, с помощью заданного выше описания структуры *point* строка

```
struct point pt;
```

определяет структурную переменную *pt* типа *struct point*. Структурную переменную при ее определении можно инициализировать, формируя список инициализаторов ее элементов в виде константных выражений:

```
struct point maxpt = {320, 200};
```

Инициализировать автоматические структуры можно также присваиванием или обращением к функции, возвращающей структуру соответствующего типа.

Доступ к отдельному элементу структуры осуществляется посредством конструкции вида:

имя-структуры.*элемент*

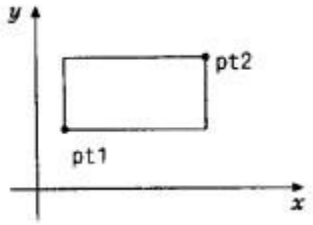
Оператор доступа к элементу структуры **.** соединяет имя структуры и имя элемента. Чтобы напечатать, например, координаты точки *pt*, годится следующее обращение к *printf*:

```
printf("%d, %d", pt.x, pt.y);
```

Другой пример: чтобы вычислить расстояние от начала координат (0,0) до *pt*, можно написать

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Структуры могут быть вложены друг в друга. Одно из возможных представлений прямоугольника - это пара точек на углах одной из его диагоналей:




```
struct rect {
   struct point pt1;
   struct point pt2;
};
```

Структура *rect* содержит две структуры *point*. Если мы объявим *screen* как

```
struct rect screen;
```

то

```
screen.pt1.x
```

обращается к координате *x* точки *pt1* из *screen*.

4.17.2 Структуры и функции

Единственно возможные операции над структурами - это их копирование, присваивание, взятие адреса с помощью & и осуществление доступа к ее элементам. Копирование и присваивание также включают в себя передачу функциям аргументов и возврат ими значений. Структуры нельзя сравнивать. Инициализировать структуру можно списком константных значений ее элементов; автоматическую структуру также можно инициализировать присваиванием.

Чтобы лучше познакомиться со структурами, напишем несколько функций, манипулирующих точками и прямоугольниками. Возникает вопрос: а как передавать функциям названные объекты? Существует по крайней мере три подхода: передавать компоненты по отдельности, передавать всю структуру целиком и передавать указатель на структуру. Каждый подход имеет свои плюсы и минусы.

Первая функция, *makepoint*, получает два целых значения и возвращает структуру *point*.

```
/* makepoint: формирует точку по компонентам x и y */
struct point makepoint(int x, int y)
{
   struct point temp;

   temp.x = x;
   temp.y = y;
   return temp;
}
```

Заметим: никакого конфликта между именем аргумента и именем элемента структуры не возникает; более того, сходство подчеркивает родство обозначаемых им объектов.

Теперь с помощью *makepoint* можно выполнять динамическую инициализацию любой структуры или формировать структурные аргументы для той или иной функции:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
            (screen.pt1.y + screen.pt2.y)/2);
```

Следующий шаг состоит в определении ряда функций, реализующих различные операции над точками. В качестве примера рассмотрим следующую функцию:

```
/* addpoint: сложение двух точек */
struct point addpoint(struct point p1, struct point p2)
{
   p1.x += p2.x;
   p1.y += p2.y;
   return p1;
}
```

Здесь оба аргумента и возвращаемое значение - структуры. Мы увеличиваем компоненты прямо в p1 и не используем для этого временной переменной, чтобы подчеркнуть, что структурные параметры передаются по значению так же, как и любые другие.

В качестве другого примера приведем функцию ptinrect, которая проверяет: находится ли точка внутри прямоугольника, относительно которого мы принимаем соглашение, что в него входят его левая и нижняя стороны, но не входят верхняя и правая.

```
/* ptinrect: возвращает 1, если p в r, и 0 в противном случае */
int ptinrect(struct point p, struct rect r)
{
   return p.x >= r.pt1.x && p.x < r.pt2.x
      && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

Здесь предполагается, что прямоугольник представлен в стандартном виде, т.е. координаты точки *pt1* меньше соответствующих координат точки *pt2*. Следующая функция гарантирует получение прямоугольника в каноническом виде.

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonrect: канонизация координат прямоугольника */
struct rect canonrect(struct rect r)
{
   struct rect temp;

   temp.pt1.x = min(r.pt1.x, r.pt2.x);
   temp.ptl.y = min(r.pt1.y, r.pt2.y);
   temp.pt2.x = max(r.pt1.x, r.pt2.x);
   temp.pt2.y = max(r.pt1.y, r.pt2.y);
   return temp;
}
```

Если функции передается большая структура, то, чем копировать ее целиком, эффективнее передать указатель на нее. Указатели на структуры ничем не отличаются от указателей на обычные переменные. Объявление

```
struct point *pp;
```

сообщает, что *pp* - это указатель на структуру типа *struct point*. Если *pp* указывает на структуру *point*, то **pp* - это сама структура, а *(*pp).x* и *(*pp).y* - ее элементы. Используя указатель *pp*, мы могли бы написать

```
struct point origin, *pp;

pp = &origin;
printf("origin: (%d,%d)\n", (*pp).x, (*pp).y);
```

Скобки в *(*pp).x* необходимы, поскольку приоритет оператора **.** выше, чем приоритет ***.** Выражение **pp.x* будет проинтерпретировано как **(pp.x)*, что неверно, поскольку *pp.x* не является указателем.

Указатели на структуры используются весьма часто, поэтому для доступа к ее элементам была придумана еще одна, более короткая форма записи. Если *p* — указатель на структуру, то

p->*элемент-структуры*

есть ее отдельный элемент. (Оператор **->** состоит из знака -, за которым сразу следует знак >.) Поэтому *printf* можно переписать в виде

```
printf("origin: (%d,%d)\n", pp->x, pp->y);
```

Операторы . и -> выполняются слева направо. Таким образом, при наличии объявления

```
struct rect r, *rp = &r;
```

следующие четыре выражения будут эквивалентны:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Операторы доступа к элементам структуры . и -> вместе с операторами вызова функции () и индексации массива [] занимают самое высокое положение в иерархии приоритетов и выполняются раньше любых других операторов. Например, если задано объявление

```
struct {
   int len;
   char *str;
```

```
} *p;
```

то

```
++p->len
```

увеличит на 1 значение элемента структуры *len*, а не указатель *p*, поскольку в этом выражении как бы неявно присутствуют скобки: *++(p->len)*. Чтобы изменить порядок выполнения операций, нужны явные скобки. Так, в *(++p)->len*, прежде чем взять значение *len*, программа прирастит указатель *p*. В *(p++)->len* указатель *p* увеличится после того, как будет взято значение *len* (в последнем случае скобки не обязательны).

По тем же правилам **p->str* обозначает содержимое объекта, на который указывает *str*; **p->str++* прирастит указатель *str* после получения значения объекта, на который он указывал (как и в выражении **s++*), *(*p->str)++* увеличит значение объекта, на который указывает *str*; **p++->str* увеличит *p* после получения того, на что указывает *str*.

4.17.3. Массивы структур

Рассмотрим программу, определяющую число вхождений каждого ключевого слова в текст Си-программы. Нам нужно уметь хранить ключевые слова в виде массива строк и счетчики ключевых слов в виде массива целых. Один из возможных вариантов - это иметь два параллельных массива:

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Однако именно тот факт, что они параллельны, подсказывает нам другую организацию хранения - через массив структур. Каждое ключевое слово можно описать парой характеристик

```
char *word;
int count;
```

Такие пары составляют массив. Объявление

```
struct key {
   char *word;
   int count;
} keytab[NKEYS];
```

объявляет структуру типа *key* и определяет массив *keytab*, каждый элемент которого является структурой этого типа и которому где-то будет выделена память. Это же можно записать и по-другому:

```
struct key {
   char *word;
   int count;
};
struct key keytab[NKEYS];
```

Так как *keytab* содержит постоянный набор имен, его легче всего сделать внешним массивом и инициализировать один раз в момент определения. Инициализация структур аналогична ранее демонстрировавшимся инициализациям - за определением следует список инициализаторов, заключенный в фигурные скобки:

```
struct key {
   char *word;
   int count;
} keytab[] = {
   "auto", 0,
   "break", 0,
   "case", 0,
   "char", 0,
   "const", 0,
   "continue", 0,
   "default", 0,
   /*...*/
   "unsigned", 0,
   "void", 0,
   "volatile", 0,
   "while", 0
};
```

Инициализаторы задаются парами, чтобы соответствовать конфигурации структуры. Строго говоря, пару инициализаторов для каждой отдельной структуры следовало бы заключить в фигурные скобки, как, например, в

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
   ...
```

Однако когда инициализаторы - простые константы или строки символов и все они имеются в наличии, во внутренних скобках нет необходимости. Число элементов массива *keytab* будет вычислено по количеству инициализаторов, поскольку они представлены полностью, а внутри квадратных скобок "[]" ничего не задано.

NKEYS - количество ключевых слов в *keytab*. Хотя мы могли бы подсчитать число таких слов вручную, гораздо легче и безопасней сделать это с помощью машины, особенно если список ключевых слов может быть изменен. Одно из возможных решений — поместить в конец списка инициализаторов пустой указатель (NULL) и затем перебирать в цикле элементы *keytab*, пока не встретится концевой элемент.

Но возможно и более простое решение. Поскольку размер массива полностью определен во время компиляции и равен произведению количества элементов массива на размер его отдельного элемента, число элементов массива можно вычислить по формуле

размер keytab / *размер* struct key

В Си имеется унарный оператор **sizeof**, который работает во время компиляции. Его можно применять для вычисления размера любого объекта. Выражения

sizeof *объект*

и

sizeof (*имя типа*)

выдают целые значения, равные размеру указанного объекта или типа в байтах. (Строго говоря, sizeof выдает беззнаковое целое, тип которого size_t определена заголовочном файле <stddef.h>.) Что касается объекта, то это может быть переменная, массив или структура. В качестве имени типа может выступать имя базового типа (int, double ...) или имя производного типа, например структуры или указателя.

В нашем случае, чтобы вычислить количество ключевых слов, размер массива надо поделить на размер одного элемента. Указанное вычисление используется в инструкции *#define* для установки значения NKEYS:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Этот же результат можно получить другим способом - поделить размер массива на размер какого-то его конкретного элемента:

```
#define NKEYS (sizeof keytab / sizeof keytab[0])
```

Преимущество такого рода записей в том, что их не надо корректировать при изменении типа.

Поскольку препроцессор не обращает внимания на имена типов, оператор **sizeof** нельзя применять в **#if**. Но в **#define** выражение препроцессором не вычисляется, так что предложенная нами запись допустима.

4.17.4. Указатели на структуры

Для иллюстрации некоторых моментов, касающихся указателей на структуры и массивов структур, воспользуемся для получения элементов массива вместо индексов указателями.

```
struct A *p;
```

Если p - это указатель на структуру, то при выполнении операций с p учитывается размер структуры. Поэтому p++ увеличит p на такую величину, чтобы выйти на следующий структурный элемент массива, а проверка условия вовремя остановит цикл.

Не следует, однако, полагать, что размер структуры равен сумме размеров ее элементов. Вследствие выравнивания объектов разной длины в структуре могут появляться безымянные "дыры". Например, если переменная типа char занимает один байт, а int - четыре байта, то для структуры

```
struct {
   char c;
   int i;
};
```

может потребоваться восемь байтов, а не пять. Оператор sizeof возвращает правильное значение.

Наконец, несколько слов относительно формата программы. Если функция возвращает значение сложного типа, как, например, в нашем случае она возвращает указатель на структуру:

```
struct A *func(…)
```

то "высмотреть" имя функции оказывается совсем не просто. В подобных случаях иногда пишут так:

```
struct A *
func(…)
```

Какой форме отдать предпочтение - дело вкуса. Выберите ту, которая больше всего вам нравится, и придерживайтесь ее.

4.17.5. Структуры со ссылками на себя

Предположим, что мы хотим решить более общую задачу - написать программу, подсчитывающую частоту встречаемости для любых слов входного потока. Так как список слов заранее не известен, мы не можем предварительно упорядочить его и применить бинарный поиск. Было бы неразумно пользоваться и линейным поиском каждого полученного слова, чтобы определять, встречалось оно ранее или нет - в этом случае программа работала бы слишком медленно. (Более точная оценка: время работы такой программы пропорционально квадрату количества слов.) Как можно организовать данные, чтобы эффективно справиться со списком произвольных слов?

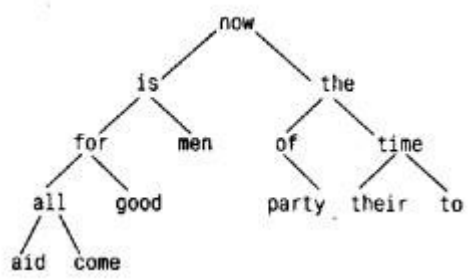
Мы воспользуемся структурой данных, называемой бинарным деревом.

В дереве на каждое отдельное слово предусмотрен "узел", который содержит:

- указатель на текст слова;
- счетчик числа встречаемости;
- указатель на левый сыновний узел;
- указатель на правый сыновний узел.

У каждого узла может быть один или два сына, или узел вообще может не иметь сыновей.

Узлы в дереве располагаются так, что по отношению к любому узлу левое поддерево содержит только те слова, которые лексикографически меньше, чем слово данного узла, а правое - слова, которые больше него. Вот как выглядит дерево, построенное для фразы "now is the time for all good men to come to the aid of their party" ("настало время всем добрым людям помочь своей партии"), по завершении процесса, в котором для каждого нового слова в него добавлялся новый узел:



Чтобы определить, помещено ли уже в дерево вновь поступившее слово, начинают с корня, сравнивая это слово со словом из корневого узла. Если они совпали, то ответ на вопрос — положительный. Если новое слово меньше слова из дерева, то поиск продолжается в левом поддереве, если больше, то — в правом. Если же в выбранном направлении поддерева не оказалось, то этого слова в дереве нет, а пустующая позиция, говорящая об отсутствии поддерева, как раз и есть то место, куда нужно "подвесить" узел с новым словом. Описанный процесс по сути рекурсивен, так как поиск в любом узле использует результат поиска в одном из своих сыновних узлов. В соответствии с этим для добавления узла и печати дерева здесь наиболее естественно применить рекурсивные функции.

Вернемся к описанию узла, которое удобно представить в виде структуры с четырьмя компонентами:

```
struct tnode {          // узел дерева
   char *word;          // указатель на текст
   int count;           // число вхождений
   struct tnode *left; // левый сын
   struct tnode *right; // правый сын
};
```

Приведенное рекурсивное определение узла может показаться рискованным, но оно правильное. Структура не может включать саму себя, но ведь

```
struct tnode *left;
```

объявляет *left* как указатель на *tnode*, а не сам *tnode*.

Иногда возникает потребность во взаимоссылающихся структурах: двух структурах, ссылающихся друг на друга. Прием, позволяющий справиться с этой задачей, демонстрируется следующим фрагментом:

```
struct t {
   ...
   struct s *p; /* p указывает на s */
};
struct s {
   ...
   struct t *q; /* q указывает на t */
```

```
}
```

Функция *addtree (добавить узел)* рекурсивна. Первое слово помещается на верхний уровень дерева (корень дерева). Каждое вновь поступившее слово сравнивается со словом узла и "погружается" или в левое, или в правое поддерево с помощью рекурсивного обращения к *addtree*. Через некоторое время это слово обязательно либо совпадет с каким-нибудь из имеющихся в дереве слов (в этом случае к счетчику будет добавлена 1), либо программа встретит пустую позицию, что послужит сигналом для создания нового узла и добавления его к дереву. Создание нового узла сопровождается тем, что *addtree* возвращает на него указатель, который вставляется в узел родителя.

```
/* addtree: добавляет узел со словом w в p или ниже него */
struct tnode *addtree(struct tnode *p, char *w)
{
   int cond;

   if (p == NULL) {  /* слово встречается впервые */
      p = talloc(); /* создается новый узел */
      p->word = strdup(w);
      p->count = 1;
      p->left = p->right = NULL;
   } else if ((cond = strcmp(w, p->word)) == 0)
      p->count++;          /* это слово уже встречалось */
   else if (cond < 0)       /* меньше корня левого поддерева */
      p->left = addtree(p->left, w);
   else                     /* больше корня правого поддерева */
      p->right = addtree(p->right, w);
   return p;
}
```

Память для нового узла запрашивается с помощью программы *talloc*, которая возвращает указатель на свободное пространство, достаточное для хранения одного узла дерева, а копирование нового слова в отдельное место памяти осуществляется с помощью *strdup*. В тот (и только в тот) момент, когда к дереву подвешивается новый узел, происходит инициализация счетчика и обнуление указателей на сыновей. Мы опустили (что неразумно) контроль ошибок, который должен выполняться при получении значений от *strdup* и *talloc*.

Практическое замечание: если дерево "несбалансировано" (что бывает, когда слова поступают не в случайном порядке), то время работы программы может сильно возрасти. Худший вариант, когда слова уже упорядочены; в этом случае затраты на вычисления будут такими же, как при линейном поиске.

Прежде чем завершить обсуждение этого примера, сделаем краткое отступление от темы и поговорим о механизме запроса памяти. Очевидно, хотелось бы иметь всего лишь одну функцию, выделяющую память, даже если эта память предназначается для разного рода объектов. Но если одна и та же функция обеспечивает память, скажем, и для указателей на *char*, и для указателей на *struct tnode*, то возникают два вопроса. Первый: как справиться с требованием большинства машин, в которых объекты определенного типа должны быть выровнены (например, *int* часто должны размещаться, начиная с четных адресов)? И второе: как объявить функцию-распределитель памяти, которая вынуждена в качестве результата возвращать указатели разных типов?

Вообще говоря, требования, касающиеся выравнивания, можно легко выполнить за счет некоторого перерасхода памяти.

Вопрос об объявлении типа таких функций, как **malloc**, является камнем преткновения в любом языке с жесткой проверкой типов. В Си вопрос решается естественным образом: **malloc** объявляется как функция, которая возвращает указатель на **void**. Полученный указатель затем явно приводится к желаемому типу. Описания *malloc* и связанных с ней функций находятся в стандартном заголовочном файле **<stdlib.h>**. Таким образом, функцию *talloc* можно записать так:

```
#include <stdlib.h>

/* talloc: создает tnode */
struct tnode *talloc(void)
{
   return (struct tnode *) malloc(sizeof(struct tnode));
}
```

Функция *strdup* просто копирует строку, указанную в аргументе, в место, полученное с помощью *malloc*:

```
char *strdup(char *s) /* делает дубликат s */
```

```
{
   char *p;

   p = (char *) malloc(strlen(s)+1); /* +1 для '\0' */
   if (p != NULL)
      strcpy(p, s);
   return p;
}
```

Функция *malloc* возвращает NULL, если свободного пространства нет; *strdup* возвращает это же значение, оставляя заботу о выходе из ошибочной ситуации вызывающей программе.

Память, полученную с помощью *malloc*, можно освободить для повторного использования, обратившись к функции *free.*

4.17.6. Средство typedef

Язык Си предоставляет средство, называемое **typedef**, которое позволяет давать типам данных новые имена. Например, объявление

```
typedef int Length;
```

делает имя *Length* синонимом *int*. С этого момента тип *Length* можно применять в объявлениях, в операторе приведения и т. д. точно так же, как тип *int*:

```
Length len, maxlen;
Length *lengths[];
```

Аналогично объявление

```
typedef char *String;
```

делает String синонимом char *, т. е. указателем на char, и правомерным будет, например, следующее его использование:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Заметим, что объявляемый в typedef тип стоит на месте имени переменной в обычном объявлении, а не сразу за словом typedef. С точки зрения синтаксиса слово typedef напоминает класс памяти - extern, static и т. д. Имена типов записаны с заглавных букв для того, чтобы они выделялись.

Для демонстрации более сложных примеров применения typedef воспользуемся этим средством при задании узлов деревьев, с которыми мы уже встречались ранее.

```
typedef struct tnode *Treeptr;

typedef struct tnode { /* узел дерева: */
   char *word;       /* указатель на текст */
   int count;        /* число вхождений */
   Treeptr left;     /* левый сын */
   Treeptr right;    /* правый сын */
} Treenode;
```

В результате создаются два новых названия типов: *Treenode* (структура) и *Treeptr* (указатель на структуру). Теперь программу *talloc* можно записать в следующем виде:

```
Treeptr talloc(void)
{
   return (Treeptr) malloc(sizeof(Treenode));
}
```

Следует подчеркнуть, что объявление *typedef* не создает объявления нового типа, оно лишь сообщает новое имя уже существующему типу. Никакого нового смысла эти новые имена не несут, они объявляют переменные в точности с теми же свойствами, как если бы те были объявлены напрямую без переименования типа. Фактически *typedef* аналогичен *#define* с тем лишь отличием, что при интерпретации компилятором он может справиться с такой текстовой подстановкой, которая не может быть обработана препроцессором. Например

```
typedef int (*PFI)(char *, char *);
```

создает тип PFI - "указатель на функцию (двух аргументов типа char *), возвращающую int.

Помимо просто эстетических соображений, для применения *typedef* существуют две важные причины. Первая - параметризация программы, связанная с проблемой переносимости. Если с помощью *typedef* объявить типы данных, которые, возможно, являются машинно-зависимыми, то при переносе программы на другую машину потребуется внести изменения только в определения *typedef*. Одна из

распространенных ситуаций - использование *typedef*-имен для варьирования целыми величинами. Для каждой конкретной машины это предполагает соответствующие установки *short*, *int* или *long*, которые делаются аналогично установкам стандартных типов, например *size_t* и *ptrdiff_t*.

Вторая причина, побуждающая к применению *typedef*,- желание сделать более ясным текст программы. Тип, названный *Treeptr* (от английских слов tree - дерево и pointer - указатель), более понятен, чем тот же тип, записанный как указатель на некоторую сложную структуру.

4.18. Объединения

Объединение - это переменная, которая может содержать (в разные моменты времени) объекты различных типов и размеров. Все требования относительно размеров и выравнивания выполняет компилятор. Объединения позволяют хранить разнородные данные в одной и той же области памяти без включения в программу машинно-зависимой информации. Эти средства аналогичны вариантным записям в Паскале.

Примером использования объединений мог бы послужить сам компилятор, заведующий таблицей символов, если предположить, что константа может иметь тип *int*, *float* или являться указателем на символ и иметь тип *char* *. Значение каждой конкретной константы должно храниться в переменной соответствующего этой константе типа. Работать с таблицей символов всегда удобнее, если значения занимают одинаковую по объёму память и запоминаются в одном и том же месте независимо от своего типа. Цель введения в программу объединения - иметь переменную, которая бы на законных основаниях хранила в себе значения нескольких типов. Синтаксис объединений аналогичен синтаксису структур. Приведем пример объединения.

```
union u_tag {
   int ival;
   float fval;
   char *sval;
} u;
```

Переменная *u* будет достаточно большой, чтобы в ней поместилась любая переменная из указанных трех типов: точный ее размер зависит от реализации. Значение одного из этих трех типов может быть присвоено переменной *u* и далее использовано в выражениях, если это правомерно, т. е. если тип взятого ею значения совпадает с типом последнего присвоенного ей значения. Выполнение этого требования в каждый текущий момент - целиком на совести программиста. В том случае, если нечто запомнено как значение одного типа, а извлекается как значение другого типа, результат зависит от реализации. Синтаксис доступа к элементам объединения следующий:

имя-объединения.элемент

или

указатель-на-объединение->элемент

т. е. в точности такой, как в структурах. Если для хранения типа текущего значения *u* использовать, скажем, переменную *utype*, то можно написать такой фрагмент программы:

```
if (utype == INT)
   printf("%d\n", u.ival);
else if (utype === FLOAT)
   printf("%f\n", u.fval);
else if (utype == STRING)
   printf("%s\n", u.sval);
else
   printf ("неверный тип %d в utype\n", utype);
```

Объединения могут входить в структуры и массивы, и наоборот. Запись доступа к элементу объединения, находящегося в структуре (как и структуры, находящейся в объединении), такая же, как и для вложенных структур. Например, в массиве структур

```
struct {
   char *name;
   int flags;
   int utype;
   union {
      int ival;
      float fval;
      char *sval;
   } u;
} symtab[NSYM];
```

к *ival* обращаются следующим образом:

symtab[i].u.ival

а к первому символу строки *sval* можно обратиться любым из следующих двух способов:

*symtab[i].u.sval

symtab[i].u.sval[0]

Фактически объединение - это структура, все элементы которой имеют нулевое смещение относительно ее базового адреса и размер которой позволяет поместиться в ней самому большому ее элементу, а выравнивание этой структуры удовлетворяет всем типам объединения. Операции, применимые к структурам, годятся и для объединений, т. е. законны присваивание объединения и копирование его как единого целого, взятие адреса от объединения и доступ к отдельным его элементам.

Инициализировать объединение можно только значением, имеющим тип его первого элемента; таким образом, упомянутую выше переменную *u* можно инициализировать лишь значением типа *int*.

4.19. Битовые поля

При дефиците памяти может возникнуть необходимость запаковать несколько объектов в одно слово машины. Одна из обычных ситуаций, встречающаяся в задачах обработки таблиц символов для компиляторов, - это объединение групп однобитовых флажков. Форматы некоторых данных могут от нас вообще не зависеть и диктоваться, например, интерфейсами с аппаратурой внешних устройств: здесь также возникает потребность адресоваться к частям слова.

Вообразим себе фрагмент компилятора, который заведует таблицей символов. Каждый идентификатор программы имеет некоторую связанную с ним информацию: например, представляет ли он собой ключевое слово и, если это переменная, к какому классу принадлежит: внешняя и/или статическая и т. д. Самый компактный способ кодирования такой информации - расположить однобитовые флажки в одном слове типа *char* или *int*.

Один из распространенных приемов работы с битами основан на определении набора "масок", соответствующих позициям этих битов, как, например, в

```
#define KEYWORD 01     /* ключевое слово */
#define EXTERNAL 02    /* внешний */
#define STATIC 04      /* статический */
```

Числа должны быть степенями двойки. Тогда доступ к битам становится делом "побитовых операций" - сдвиг, маскирование, взятие дополнения. Некоторые виды записи выражений встречаются довольно часто. Так,

```
flags |= EXTERNAL | STATIC;
```

устанавливает 1 в соответствующих битах переменной *flags*,

```
flags &= ~(EXTERNAL | STATIC);
```

обнуляет их, а

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

оценивает условие как истинное, если оба бита нулевые.

Хотя научиться писать такого рода выражения не составляет труда, вместо побитовых логических операций можно пользоваться предоставляемым Си другим способом прямого определения и доступа к полям внутри слова. Битовое поле (или для краткости просто поле) - это некоторое множество битов, лежащих рядом внутри одной, зависящей от реализации единицы памяти, которую мы будем называть "словом". Синтаксис определения полей и доступа к ним базируется на синтаксисе структур. Например, строки *#define*, фигурировавшие выше при задании таблицы символов, можно заменить на определение трех полей:

```
struct {
   unsigned int is_keyword : 1;
   unsigned int is_extern  : 1;
   unsigned int is_static  : 1;
} flags;
```

Эта запись определяет переменную *flags*, которая содержит три однобитовых поля. Число, следующее за двоеточием, задает ширину поля. Поля объявлены как *unsigned int*, чтобы они воспринимались как беззнаковые величины.

На отдельные поля ссылаются так же, как и на элементы обычных структур: *flags.is_keyword*, *flags.is_extern* и т.д. Поля "ведут себя" как малые целые и могут участвовать в арифметических выражениях точно так же, как и другие целые. Таким образом, предыдущие примеры можно написать более естественно:

```
flags.is_extern = flags.is_static = 1;
```

устанавливает 1 в соответствующие биты;

```
flags.is_extern = flags.is_static = 0;
```

их обнуляет, а

```
if (flags.is_extern == 0 && flags.is_ststic == 0)
   ...
```

проверяет их.

Почти все технические детали, касающиеся полей, в частности, возможность поля перейти границу слова, зависят от реализации. Поля могут не иметь имени; с помощью безымянного поля (задаваемого только двоеточием и шириной) организуется пропуск нужного количества разрядов. Особая ширина, равная нулю, используется, когда требуется выйти на границу следующего слова.

На одних машинах поля размещаются слева направо, на других - справа налево. Это значит, что при всей полезности работы с ними, если формат данных, с которыми мы имеем дело, дан нам свыше, то необходимо самым тщательным образом исследовать порядок расположения полей; программы, зависящие от такого рода вещей, не переносимы. Поля можно определять только с типом *int*, а для того чтобы обеспечить переносимость, надо явно указывать *signed* или *unsigned*. Они не могут быть массивами и не имеют адресов, и, следовательно, оператор **&** к ним не применим.

4.20. ГРАФИЧЕСКИЕ ПРИМИТИВЫ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

На большинстве ЭВМ (включая и 1BM PC/AT) принят растровый способ изображения графической информации - изображение представлено прямоугольной матрицей точек (пикселов), и каждый пиксел имеет свой цвет, выбираемый из заданного набора цветов - палитры. Для реализации этого подхода компьютер содержит в своем составе видеоадаптер, который, с одной стороны, хранил в своей памяти (ее принято называть видеопамятью) изображение (при этом на каждый ппксел изображения отводится фиксированное количество бит памяти), а с другой - обеспечивает регулярное (50-70 раз в секунду) отображение видеопамяти на экране монитора. Размер палитры определяется объемом видеопамяти, отводимой под один пиксел, и зависит от типа видеоадаптера.

Для ПЭВМ типа 1BM PC/AT и PS/2 существует несколько различных типов видеоадаптеров, различающихся как своими возможностями, так и аппаратным устройством и принципами работы с ними. Основными видеоадаптерами для этих машин являются CGA, EGA, VGA и Hercules. Существует также большое количество адаптеров, совместимых с EGA/VGA, но предоставляюших по сравнению с ними ряд дополнительных возможностей.

Практически каждый видеоадаптер поддерживает несколько режимов работы, отчичаюшихся друг от друга размерами матрицы пикселов (разрешением) и размером палитры (количеством цветов, которые можно одновременно отобразить на экране). Зачастую разные режимы даже одного адаптера имеют разную организацию видеопамяти и способы работы с ней. Более подробную информацию о работе с видеоадаптерами можно получить из следуюшей главы.

Однако большинство адаптеров строится по принципу совместимости с предыдущими. Так, адаптер EGA поддерживает все режимы адаптера CGA. Поэтому любая программа, рассчитанная на работу с адаптером CGA, будет также работать и с адаптером EGA, даже не замечая этого. При этом адаптер EGA поддерживает, конечно, еще ряд своих собственных режимов. Аналогично адаптер VGA поддерживает все режимы адаптера EGA.

Фактически любая графическая операция сводится к работе с отдельными пикселами - поставить точку заданного цвета и узнать цвет заданной точки. Однако большинство графических библиотек поддерживают работу и с более сложными объектами, поскольку работа только на уровне отдельно взятых пикселов была бы очень затруднительной для программиста и, к тому же, неэффективной.

Среди подобных объектов (представляющих собой объединения пикселов) можно выделить следующие основные группы:

- линейные изображения (растровые образы линий);
- сплошные объекты (растровые образы двумерных областей);
- шрифты;
- изображения (прямоугольные матрицы пикселов).

Как правило, каждый компилятор имеет свою графическую библиотеку, обеспечивающую работу с основными группами графических объектов. При этом требуется, чтобы подобная библиотека поддерживала работу с основными типами видеоадаптеров.

Существует несколько путей обеспечения этого.

Один из них заключается в написании версий библиотеки для всех основных типов адаптеров. Однако программист должен изначально знать, для какого конкретно видеоадаптера он пишет свою программу, и использовать соответствующую библиотеку. Полученная при этом программа уже не будет работать на других адаптерах, несовместимых с тем, для которого писалась программа. Поэтому вместо одной программы получается целый набор программ для разных видеоадаптеров. Принцип совместимости адаптеров выручает здесь несильно: хотя программа, рассчитанная на адаптер CGA, и будет работать на

VGA, но она не сможет полностью использовать все его возможности и будет работать с ним только как с CGA.

Можно включить в библиотеку версии процедур для всех основных типов адаптеров. Это обеспечит некоторую степень машинной независимости. Однако нельзя исключать случай наличия у пользователя программы какого-либо типа адаптера, не поддерживаемого библиотекой (например, SVGA). Но самым существенным недостатком такого подхода является слишком большой размер получаемого выполняемого файла, что уменьшает объем оперативной памяти, доступный пользователю.

Наиболее распространенным является использование драйверов устройств. При этом выделяется некоторый основной набор графических операций так, что все остальные операции можно реализовать, используя только операции основного набора. Привязка к видеоадаптеру заключается именно в реализации этих основных (базисных) операций. Для каждого адаптера пишется так называемый драйвер - небольшая программа со стандартным интерфейсом, реализующая все эти операции для данного адаптера и помешаемая в отдельный файл. Библиотека в начале своей работы определяет тип имеющегося видеоадаптера и загружает соответствуюший драйвер в память. Таким образом достигается почти полная машинная независимость написанных программ.

Рассмотрим работу одной из наиболее популярных графических библиотек - библиотеки компилятора Borland C++. Для использования этой библиотеки необходимо сначала подключить ее при помощи команды меню Options/Linker/Libraries.

Рассмотрим основные группы операций.

4.20.1. Инициализация и завершение работы с библиотекой

Для инициализации библиотеки служит функция

void far initgraph (int far *drive, int far «mode. char far *path);

Первый параметр задает библиотеке тип адаптера, с которым будет вестись работа. В соответствии с этим параметром будет загружен драйвер указанного видеоадаптера и произведена инициализация всей библиотеки. Определен ряд констант, задающих набор стандартных драйверов: CGA, EGA. VGA, DETECT и другие.

Значение DETECT сообщает библиотеке о том, что тип имеющегося видеоадаптера надо определить ей самой и выбрать для него режим наибольшего разрешения.

Второй параметр - mode - определяет режим.

Параметр	Режим
CGACO, CGACI, COAC2, CGAC3	320 на 200 точек на 4 цвета
CGAHI	640 на 200 точек на 2 цвета
EGALO	640 на 200 точек на 16 цветов
EGAHI	640 на 350 точек на 16 цветов
VGALO	640 на 200 точек на 16 цветов
VGAMED	640 на 350 точек на 16 цветов
VGAHI	640 на 4SO точек на 16 цветов

Если в качестве первого параметра было взято значение DETECT, то параметр mode не используется.

В качестве третьего параметра выступает имя каталога, где находится драйвер адаптера - файл типа BGI (Borland's Graphics Interface):

CGA.BGl - драйвер адаптера CGA;

EGAVGA.BGI- драйвер адаптеров EGA и VGA;

HERC.BGI - драйвер адаптера Hercules.

Функция graphresult возвращает код завершения предыдущей графической операции

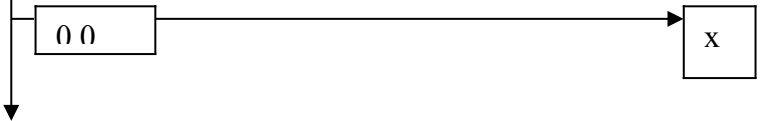
int far graphresult (void);

Успешному выполнению соответствует значение grOk. В случае ошибки выдается стандартное диагностическое сообщение.

Для окончания работы с библиотекой необходимо вызвать функцию closegraph:

Void far closegraph()

После инициализации библиотеки адаптер переходит в соответствующий режим, экран очищается и на нем устанавливается следующая координатная система. Начальная точка с координатами (О, 0) располагается в левом верхнем углу экрана.



Узнать максимальные значения X и Y координат пиксела можно, используя функции getmaxx и getmaxy:

int far getmaxx (void);

int far getmaxy (void.);

Узнать, какой именно режим в действительности установлен, можно при помощи функции getgraphmode:

int far getgraphmode (void);

Для очистки экрана удобно использовать функцию clearviewport:

void far clearvievport (void);

4.20.2. Работа с отдельными точками

Функция putpixel ставит пиксел заданного цвета Color в точке с координатами (x, y):

void far putpixel (int x, int y, int Color);

Функция getplXel возвращает цвет пиксела с координатами (x, y):

unsigned far getpixel (int x, int y);

4.20.3. Рисование линейных объектов

При рисовании линейных объектов основным инструментом является перо, которым эти объекты рисуются. Перо имеет следующие характеристики:

1. цвет (по умолчанию белый);
2. толщина '(по умолчанию 1);
3. шаблон (по умолчанию сплошной).

Шаблон служит для рисования пунктирных и штрихпунктирных линий. Для установки параметров пера используются следующие функции выбора.

Процедура setcolor устанавливает цвет пера:

void far setcolor (int Color);

Функция setlinestyle определяет остальные параметры пера:

void far setlinestyle (int Style, unsigned Pattern, int Thickness);

Первый параметр задает шаблон линии. Обычно в качестве этого параметра выступает один из предопределенных шаблонов: SOLID LINE, DOTTED LINE, CENTERLINE, DASHED LINE, USERBIT LINE и другие. Значение USERBIT LINE означает, что шаблон задается (пользователем) вторым параметром. Шаблон определяется 8 битами, где значение бита 1 означает, что в соответствующем месте будет поставлена точка, а значение 0 - что точка ставиться не, будет.

Третий параметр задает толщину линии в пикселах. Возможные значения параметра – NORM_WIDTH и THICK_WIDTH (1 и 3). При помоши пера можно рисовать ряд линейных объектов-прямолинейные отрезки, дуги окружностей и эллипсов, ломаные.

4.20.3.1. Рисование прямолинейных отрезков

Функция line рисует отрезок, соединяющий точки (x_1, y_1) и (x_2, y_2):

void far line (int x1; int .y1, int x2, int y2)

4.20.3.2. Рисование окружностей

Функция circle рисует окружность радиуса r с центром в точке (x, y):

void far circle (int x, int y, int r);

4.20.3.3. Рисование дуг эллипса

Функции arc и ellipse рисуют дуги окружности (с центром в точке (x, y) и радиусом r) и эллипса (с центром (x, y), полуосями rx и ry, параллельными координатным осям), начиная с угла StartAngle и заканчивая углом EndAngle.

Углы задаются в градусах в направлении против часовой стрелки:

void far arc (int x, int y, int StartAngle, .int EndAng1e, int r);

void far ellipse (int x, int y, int StartAngle, int EndAngle, int rx, int ry);

4.20.4. Рисование сплошных объектов

4.20.4.1. Закрашивание объектов

С понятием закрашивания тесно связано понятие кисти. Кисть определяется цветом и шаблоном - матрицей 8 на 8 точек (бит), где бит, равный 1, означает, что нужно ставить точку цвета кисти, а 0 что нужно ставить черную точку (цвета 0).

Для задания кисти используются следующие функции:

void far setfillstyle(int Pattern, int Color);

void far setfillpattern (char far Pattern, int Color);

Функция setfillstyle служит для задания кисти. Параметр Style определяет шаблон кисти либо как один из стандартных (EMPTY FILL, SOLID FILL, LINE FILL, LTSLASH_FILL), либо как шаблон, задаваемый пользователем (USERFILL). Пользовательский шаблон устанавливает процедура setfillpattern, первый параметр в которой и задает шаблон - матрицу 8 на 8 бит, собранных по горизонтали в байты. По умолчанию используется сплошная кисть (SOLID FILL) белого цвета.

Процедура Ьar закрашивает выбранной кистью прямоугольник с левым верхним углом (x1,y1) и правым нижним углом {x2,y2):

void far Ьar (int x1, int y1, int x2, int y2);

Функция fillellipse закрашивает сектор эллипса:

void far fillellipse (int x, int y, int StartAngle, int EndAngle, int rx, int ry);

Функция floodfill служит для закраски связной области, ограниченной линией цвета BorderColor и содержащей точку (x, y) внутри себя:

void far floodfill (int.x, int y, int BorderCo1oг);

Функция fillpoly осуществляет закраску многоугольника, заданного массивом значений x- и y-координат:

void far fillpoly (int numpoints, int far * points);

4.20.5. Работа с изображениями

Библиотека поддерживает также возможность запоминания прямоугольного фрагмента изображения в обычной (оперативной) памяти и выводе его на экран. Это может использоваться для запоминания изображения в файл, создания мультипликации и т. п.

Объем памяти, требуемый для запоминания фрагмента изображения, в байтах можно получить при помощи функции imagesize:

unsigned far imagesize (int x1, int y1, int x2, int y2);

Для запоминания изображания служит процедура getimage:

void far getimage (int x1, int y1, int x2, int y2, void far - Image);

При этом прямоугольный фрагмент, определяемый точками (x1,y1) и (x2,y2), записывается в область памяти, задаваемую последним параметром - Image.

Для вывода изображения служит процедура puttmage:

void far putimage (int x, int y, void far * Image, int op);

Хранящееся в памяти изображение, которое задается параметром Image, выводится на экран так, чтобы точка (x, y) была верхним левым углом изображения. Последний параметр определяет способ наложения выводимого изображения на уже имеющееся на экране (см.функцию setwritemode). Поскольку значение (цвет) каждого пиксела представлено фиксированным количеством бит, то в качестве возможных вариантов наложения выступают побитовые логические операции. Возможные значения для параметра op приведены ниже:

- COPY PUT - происходит простой вывод (замещение);
- NOT PUT - происходит вывод инверсного изображения;
- OR PUT - используется побитовая операция ИЛИ;
- XOR PUT - используется побитовая операция ИСКЛКЛЮЧАЮЩЕЕ ИЛИ;
- AND PUT - используется побитовая операция И.

```
unsigned ImageSize = imagesize ( x1, y1, x2, y2 );
void *Image = malloc (ImageSize);
if ( Image != NULL ) {
   getimage ( x1, y1, x2, y2, Image );
   putimage ( x, y, Image, COPY_PUT );
   free ( Image );
}
```

В этой программе происходит динамическое выделение под заданный фрагмент изображения на экране требуемого объема памяти.

Этот фрагмент запоминается в отведенную память. Далее сохраненное изображение выводится на новое место (в вершину левого верхнего угла - (x, y) и отведенная под изображение память освобождается.

4.20.6. Работа со шрифтами

Под шрифтом обычно понимается набор изображений символов. Шрифты могут различаться по организации (растровые и векторные), по размеру, по направлению вывода и по ряду других параметров. Шрифт может быть фиксированным (размеры всех символов совпадают) или пропорциональным (высоты символов совпадают, но они могут иметь разную ширину).

Для выбора шрифта и его параметров служит функция settextstyle:

void far settextstyle (int Font, int Direction, int Size);

Здесь параметр Font задает идентификатор одного из шрифтов:

- DEFAULT_FONT - стандартный растровый шрифт размером 8 на 8 точек, находящийся в ПЗУ видеоадаптера;
- TRIPLEX_FONT, GOTHIC_FONT, SANS_SERIF_FONT, SMALL_FONT - стандартные пропорциональные векторные шрифты, входящие в комплект Borland C++ (шрифты хранятся в файлах типа CHR и по этой команде подгружаются в оперативную память; файлы должны находиться в том же каталоге, что и драйверы устройств).

Параметр Direction задает направление вывода:

- HORIZ_DIR - вывод по горизонтали;
- VERT_DIR - вывод по вертикали.

Параметр Size задает, во сколько раз нужно увеличить шрифт перед выводом на экран. Допустимые значения 1, 2, ..., 10.

При желании можно использовать любые шрифты в формате CHR. Для этого надо сначала загрузить шрифт при помощи функции:

int far installuserfont (char far * FontFileNase);

а затем возвращенное функцией значение передать settextstyle в качестве идентификатора шрифта:

int MyFont = installuserfont ("MYFONT.CHR");

settextstyle (MyFont, HORIZ_DIR, 5),

Для вывода текста служит функция outtextxy:

void far outtextxy (int x, int y, char far *text);

При этом строка text выводится так, что точка (x, y) оказывается вершиной левого верхнего угла первого символа.

Для определения размера, который займет на экране строка текста при выводе текущим шрифтом, используются функции, возвращаюшие ширину и высоту в пикселах строки текста:

int far textwidth (char far * text)';

int far textheight (char far * text);

4.20.7. Понятие режима (способа) вывода

При выводе изображения на экран обычно происходит замещение пиксела, ранее находившегося на этом месте, на новый. Можно, однако, установить такой режим, что в видеопамять будет записываться результат наложения ранее имевшегося значения на выводимое.

Поскольку каждый пиксел представлен фиксированным количеством бит, то естественно, что в качестве такого наложения выступают побитовые операции. Для установки используемой операции служит процедура setwritemode:

void far setwritemode (int Mode);

Параметр Mode задает способ наложения и может принимать одно из следукнцих значений:

COPY PUT- происходит простой вывод (замещение);

XOR PUT — используется побитовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ.

Режим XOR PUT удобен тем, что повторный вывод одного и того же изображения на то же место уничтожает результат первого вывода, восстанавливая изображение, которое было на экране до этого.

Замечание.

Не все функции графической библиотеки поддерживают использование режимов вывода; например, функции закраски игнорируют установленный режим наложения (вывода). Кроме того, некоторые функции могут не совсем корректно работать в режиме XOR PUT.

4.20.8. Понятие окна (порта вывода)

При желании пользователь может создать на экране окно – своего рода маленький экран со своей локальной системой координат. Для этого служит функция setviewport:

void far setviewport (int x1, int y1, int x2, int y2, int Clip);

Эта функция устанавливает окно с глобальными координатами (x1 y1), (x2 y2) При этом локальная система координат вводится так, что точке с координатами (0, 0) соответствует точка с глобальными координатами (x1,y1). Это означает, что локальные координаты отличаются от глобальных координат лишь сдвигом на (x1,y1), причем все процедуры рисования (кроме SetViewPort) работают всегда с локальными координатами. Параметр Clip определяет, нужно ли проводить отсечение изображения, не помещающегося внутрь окна, или нет.

Замечание

Отсечение ряда обьектов проводится не совсем корректно; так, функция outtextxy производит отсечение не на уровне пикселов, а по символам.

4.20.9. Понятие палитры

Адаптер EGA и все совместимые с ним адаптеры предоставляют дополнительные возможности по управлению цветом. Наиболее распространенной схемой представления цветов для видеоустройств является так называемое RGB-представление, в котором любой цвет представляется как сумма трех основных цветов - красного (Red), зеленого (Green) и синего (Blue) с заданными интенсивностями. Все возможное пространство цветов представляет из себя единичный куб, и каждый цвет определяется тройкой чисел (r, g, b). Например желтый цвет задается как (1, 1, 0), а малиновый – как (1, 0, 1). Белому цвету соответствует набор (1, 1, 1), а черному - (0, 0, 0).

Обычно под хранение каждой из компонент цвета отводится фиксированное количество n бит памяти. Поэтому считается, что допустимый диапазон значений для компонент цвета не [0, 1], а $[0,2^n- 1]$

Практически любой видеоадаптер способен отобразить значительно большее количество цветов, чем определяется количеством бит, отводимых в видеопамяти под один пиксел. Для использования этой возможности вводится понятие палитры;

Палитра - это массив, в котором каждом) возможному значению пиксела сопоставляется значение цвета (r, g, Ь), выводимое на экран. Размер палитры и ее организация зависят от типа используемого видеоадаптера.

Наиболее простой является организация палитры на EGA-адаптере, Под каждый из 16 возможных логических цветов (значений пиксела) отводится 6 бит, по 2 бита на каждую цветовую компоненту.

При этом цвет в палитре задается байтом следующего вида: 00rgbRGB, где r, g, Ь, R, G, B могут принимать значение 0 или 1.

Используя функцию setpalette-

void far setpalette (int Color, int ColorVaIue);

можно для любого из 16 логических цветов задать .любой из 64 возможных физических цветов.

Функция getpalette-

void far getpalette (struct palettetype far * palette);

служит для получения текущей палитры, которая возвращается в виде следующей структуры:

```
struct palettetype
{
        unsigned char size;
        signed char colors[MAXCOLORS+1]
}
```

4.20.10. Понятие видеостраниц и работа с ними

Для большинства режимов (например, для EGAHI) объем видеопамяти, необходимый для хранения всего изображения (экрана), составляет менее половины имеющейся видеопамяти (256 Кбайт для EGA и VGA). В этом случае вся видеопамять делится на равные части (их количество обычно является степенью двух), называемые страницами, так, что для хранения всего изображения достаточно одной из страниц. Для режима EGAHI видеопамять делится на две страницы: 0-ю (адрес 0xA000:0) и 1-ю (адрес 0xA000: 0x8000).

Видеоадаптер отображает на экран только одну из имеющихся у него страниц. Эта страница называется видимой и устанавливается следующей процедурой setvisualpage:

void.far setvisualpage (int Page);

где Page - номер той страницы, которая станет видимой на экране после вызова этой процедуры.

Графическая библиотека также может осущесгвлять работу с любой из имеющихся страниц. Страница, с которой работает библиотека, называется активной. Активная страница устанавливается процедурой setactivepage:

void far setactivepage (int Page);

где Page - номер страницы, с которой работает библиотека и на которую происходит весь вывод.

Использование видеостраниц играет очень большую роль при мультипликации.

Реализация мультипликации на ПЭВМ заключается в последовательном рисовании на экране очередного кадра. При традиционном способе работы (кадр рисуется, экран очищается, рисуется следующий кадр) постоянные очистки экрана и построение нового изображения на чистом экране создают нежелательный эффект мерцания.

Для устранения этого эффекта очень удобно использовать страницы видеопамяти. При этом, пока на видимой странице пользователь видит один кадр, активная, но невидимая страница очищается и на ней рисуется новый кадр. Как только кадр готов, активная и видимая страницы меняются местами и пользователь вместо старого кадра сразу видит новый.

4.20.11. 16-цветные режимы адаптеров EGA и VGA

Для 16-цветных режимов под каждый пиксел изображения необходимо выделить 4 бита видеопамяти ($2^4 = 16$). Однако эти 4 бита выделяются не последовательно в одном байте, а разнесены в 4 разных блока (цветовые плоскости) видеопамяти.

Вся видеопамять карты (обычно 256 Кбайт) делится на 4 равные части, найываемые цветовыми плоскостями. Каждому пикселу ставится в соответствие по. одному биту в каждой плоскости, причем все эти биты одинаково расположены относительно ее начала. Обычно эти, плоскости представляют параллельно расположенными одна над другой, так что каждому пикселу соответствует 4 расположенных друг под другом бита. Все эти плоскости проектируются на один и тот же участок адресного пространства процессора, начиная с адреса 0xA000:0. При этом все операции чтения и записи видеопамяти опосредуются видеокартой! Поэтому, если вы записали байт по адресу 0xA000:0, то это вовсе не означает, что посланный байт в действительности запишется хотя бы в одну из этих плоскостей, точно так же как при операции чтения прочитанный байт не обязательно будет совпадать с одним из 4 байтов в соответствующих плоскостях. Механизм этого опосредования определяется логикой карты, но для программиста существует возможность известного управления этой логикой (при работе одновременно с 8 пикселами).

Для работы с пикселом необходимо определить адрес байта в видеопамяти, содержащего данный пиксел, и позицию пиксела внутри байта (поскольку один пиксел отображается на один бит в каждой плоскости, то байт соответствует сразу 8 пикселам).

Поскольку видеопамять под пикселы отводится последовательно слева направо и сверху вниз, то одна строка соответствует 80 байтам адреса и каждым 8 последовательным пикселам, начинающимся с позиции, кратной 8, соответствует один байт. Тем самым адрес байта задается выражением 80*y+(x»3), а его номер внутри байта задается выражением x&7, где (x, y) - координаты пиксела.

Для идентификации позиции пиксела внутри байта часто используется не номер бита, а битовая маска - байт, в котором отличен от нуля только бит, стоящий на позиции пиксела.

Битовая маска задается следующим выражением. 0x80»(x&7).

На видеокарте находится набор специальных 8-битовых регистров. Часть из них доступна только для чтения, часть - только для записи, а некоторые вообще недоступны программисту. Доступ к регистрам осуществляется через порты ввода/вывода процессора.

Регистры видеокарты делятся на несколько групп. При этом каждой группе соответствует пара последовательных портов (порт адреса и порт значения). Для записи значения в реестр видеокарты необходимо сначала записать номер регистра в первый порт (порт адреса), а затем записать значение в следующий порт (порт значения). Для чтения регистра в порт адреса записывается номер регистра, а затем его значение читается из порта значения.

4.21. ПРЕОБРАЗОВАНИЯ НА ПЛОСКОСТИ

Вывод изображения на экран дисплея и разнообразные действия с ним, в том числе и визуальный анализ, требуют от пользователя известной геометрической грамотности. Геометрические понятия, формулы и факты, относящиеся прежде всего к плоскому и трехмерному случаям, играют в задачах компьютерной графики особую роль. Геометрические соображения, подходы и идеи в соединении с постоянно расширяющимися возможностями вычислительной техники являются неиссякаемым источником существенных продвижений на пути развития компьютерной графики, ее эффективного использования в научных и иных исследованиях. Порой даже самые простые геометрические методики обеспечивают заметные продвижения на отдельных этапах решения большой графической задачи.

Заметим прежде всего, что особенности использования геометрических понятий, формул и фактов, как простых и хорошо известных, так и новых более сложных, требуют особого взгляда на них и иного осмысления.

4.21.1. Аффинные преобразования на плоскости

В компьютерной графике все, что относится к двумерному случаю, принято обозначать символом (2D) (2-dimension).

Допустим, на плоскости введена прямолинейная координатная система. Тогда каждой точке М ставится в соответствие упорядоченная пара чисел (x, y) ее координат. Вводя на плоскости еще одну прямолинейную систему координат, мы ставим в соответствие той же точке М другую пару чисел - (x*, y*).

Переход от одной прямолинейной координатной системы на плоскости к другой описывается следующими соотношениями

$$x^* = \alpha x + \beta y + \lambda$$
$$y^* = \gamma x + \delta y + \mu, \qquad (*)$$
$$\text{ïðè÷åì} \quad \begin{vmatrix} \alpha & \beta \\ \gamma & \delta \end{vmatrix} \neq 0$$

В дальнейшем мы будем рассматривать формулы (*) как правило, согласно которому в заданной системе прямолинейных координат преобразуются точки плоскости.

В аффинных преобразованиях плоскости особую роль играют несколько важных частных случаев, имеющих хорошо прослеживаемые геометрические характеристики. При исследовании геометрического смысла числовых коэффициентов в формулах (*) для этих случаев нам удобно считать, что заданная система координат является прямоугольной декартовой.

А. Поворот (вокруг начальной точки на угол φ описывается формулами

$$x^* = x\cos\varphi - y\sin\varphi$$
$$y^* = x\sin\varphi + y\cos\varphi$$

Б. Растяжение (сжатие) вдоль координатных осей можно задать так:

$$x^* = \alpha x,$$
$$y^* = \delta y,$$

$\text{ïðè÷åì}\ , \text{åñëè}\ \alpha > 1, \delta > 1, \text{òî}\ \text{ðàñòÿæåíèå}\ , \text{åñëè}\ \alpha < 1, \delta < 1, \text{òî}\ \text{ñæàòèå}$

В. Отражение (относительно оси абсцисс) задается при помощи формул

$$x^* = x$$
$$y^* = -y$$

Г. Пусть вектор переноса имеет координаты λ и μ. Перенос обеспечивают соотношения

$$x^* = x + \lambda$$
$$y^* = y + \mu$$

Выбор этих четырех частных случаев определяется двумя обстоятельствами.

1. Каждое из приведенных выше преобразований имеет простой и наглядный геометрический смысл (геометрическим смыслом наделены и постоянные числа, входящие в приведенные формулы).

2. Как доказывается в курсе аналитической геометрии, любое преобразование вида (') всегда можно представить как последовательное исполнение (суперпозицию) простейших преобразований вида А, Б, В и Г (или части этих преобразований).

Таким образом, справедливо следующее важное свойство аффинных преобразований плоскости: любое отображение вида (*) можно описать при помощи отображений, задаваемых формулами А, Б, В и Г.

Для эффективного использования этих известных формул в задачах компьютерной графики более удобной является их матричная запись. Матрицы, соответствующие случаям А, Б и В, строятся легко и имеют соответственно следующий вид:

$$\begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix}, \quad \begin{pmatrix} \alpha & 0 \\ 0 & \delta \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Однако для решения рассматриваемых далее задач весьма желательно охватить матричным подходом все четыре простейших преобразования (в том числе и перенос), а, значит, и обшее аффинное преобразование. Этого можно достичь, например, так: перейти к описанию произвольной точки плоскости не упорядоченной парой чисел, как это было сделано выше, а упорядоченной тройкой чисел.

Элементы произвольной матрицы аффинного преобразования не несут в себе явно выраженного геометрического смысла. Поэтому чтобы реализовать.то или иное отображение, то есть найти элементы соответствующей матрицы по заданному геометрическому описанию, необходимы специальные приемы. Обычно построение этой матрицы в соответствии со сложностью рассматриваемой задачи и с описанными выше частными случаями разбивают на несколько этапов.

На каждом этапе ищется матрица, соответствующая тому или иному из выделенных выше случаев А, Б, В или Г, обладающих хорошо выраженными геометрическими свойствами.

Выпишем соответствующие матрицы третьего порядка.

А. Матрица вращения (rotation)

$$\begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Б. Матрица растяжения(сжатия) (dilatation)

$$\begin{pmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

В. Матрица отражения (reflection)

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Г. Матрица переноса (translation)

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \lambda & \mu & 1 \end{pmatrix}$$

4.22. Доступ к файлам

Во всех предыдущих примерах мы имели дело со стандартным вводом и стандартным выводом, которые для программы автоматически предопределены операционной системой конкретной машины.

Следующий шаг - научиться писать программы, которые имели бы доступ к файлам, заранее не подсоединенным к программам. Одна из программ, в которой возникает такая необходимость, - это программа *cat*, объединяющая несколько именованных файлов и направляющая результат в стандартный вывод.

Возникает вопрос: что надо сделать, чтобы именованные файлы можно было читать; иначе говоря, как связать внешние имена, придуманные пользователем, с инструкциями чтения данных?

На этот счет имеются простые правила. Для того чтобы можно было читать из файла или писать в файл, он должен быть предварительно открыт с помощью библиотечной функции **fopen**. Функция *fopen* получает внешнее имя после чего осуществляет некоторые организационные действия и "переговоры" с операционной системой (технические детали которых здесь не рассматриваются) и возвращает указатель, используемый в дальнейшем для доступа к файлу.

Этот указатель, называемый *указателем файла*, ссылается на структуру, содержащую информацию о файле (адрес буфера, положение текущего символа в буфере, открыт файл на чтение или на запись, были ли ошибки при работе с файлом и не встретился ли конец файла). Пользователю не нужно знать подробности, поскольку определения, полученные из <stdio.h>, включают описание такой структуры, называемой **FILE**.

Единственное, что требуется для определения указателя файла, - это задать описания такого, например, вида:

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

Это говорит, что *fp* есть указатель на *FILE*, а *fopen* возвращает указатель на *FILE*. Заметим, что *FILE* — это имя типа) наподобие *int*, а не тег структуры. Оно определено с помощью **typedef**.

Обращение к *fopen* в программе может выглядеть следующим образом:

```
fp = fopen(name, mode);
```

Первый аргумент - строка, содержащая имя файла. Второй аргумент несет информацию о режиме. Это тоже строка: в ней указывается, каким образом пользователь намерен применять файл. Возможны следующие режимы: чтение (*read* - "r"), запись (*write* - "w") и добавление (*append* - "a"), т. е. запись информации в конец уже существующего файла. В некоторых системах различаются текстовые и бинарные файлы; в случае последних в строку режима необходимо добавить букву "b" (*binary* - бинарный).

Тот факт, что некий файл, которого раньше не было, открывается на запись или добавление, означает, что он создается (если такая процедура физически возможна). Открытие уже существующего файла на запись приводит к выбрасыванию его старого содержимого, в то время как при открытии файла на добавление его старое содержимое сохраняется. Попытка читать несуществующий файл является ошибкой. Могут иметь место и другие ошибки; например, ошибкой считается попытка чтения файла, который по статусу запрещено читать. При наличии любой ошибки *fopen* возвращает NULL.

Следующее, что нам необходимо знать, - это как читать из файла или писать в файл, коль скоро он открыт. Существует несколько способов сделать это, из которых самый простой состоит в том, чтобы

воспользоваться функциями **getc** и **putc**. Функция *getc* возвращает следующий символ из файла; ей необходимо сообщить указатель файла, чтобы она знала откуда брать символ.

int getc(FILE *fp);

Функция *getc* возвращает следующий символ из потока, на который указывает **fp*; в случае исчерпания файла или ошибки она возвращает EOF.

Функция *putc* пишет символ *c* в файл *fp*

int putc(int c, FILE *fp);

и возвращает записанный символ или EOF в случае ошибки. Аналогично *getchar* и *putchar*, реализация *getc* и *putc* может быть выполнена в виде макросов, а не функций.

При запуске Си-программы операционная система всегда открывает три файла и обеспечивает три файловые ссылки на них. Этими файлами являются: стандартный ввод, стандартный вывод и стандартный файл ошибок; соответствующие им указатели называются **stdin**, **stdout** и **stderr**; они описаны в <stdio.h>. Обычно *stdin* соотнесен с клавиатурой, а *stdout* и *stderr* - с экраном. Однако *stdin* и *stdout* можно связать с файлами или, используя конвейерный механизм, соединить напрямую с другими программами

С помощью *getc*, *putc*, *stdin* и *stdout* функции *getchar* и *putchar* теперь можно определить следующим образом:

#define getchar() getc(stdin)

#define putchar(c) putc((c), stdout)

Форматный ввод-вывод файлов можно построить на функциях **fscanf** и **fprintf**. Они идентичны *scanf* и *printf* с той лишь разницей, что первым их аргументом является указатель на файл, для которого осуществляется ввод-вывод, формат же указывается вторым аргументом.

int fscanf(FILE *fp, char *format, ...)

int fprintf(FILE *fp, char *format, ...)

Вот теперь мы располагаем теми сведениями, которые достаточны для написания программы *cat*, предназначенной для конкатенации (последовательного соединения) файлов. Предлагаемая версия функции *cat*, как оказалось, удобна для многих программ. Если в командной строке присутствуют аргументы, они рассматриваются как имена последовательно обрабатываемых файлов. Если аргументов нет, то обработке подвергается стандартный ввод.

```
#include <stdio.h>
/* cat: конкатенация файлов, версия 1 */
main(int argc, char *argv[])
{
   FILE *fp;
   void filecopy(FILE *, FILE *);

   if (argc == 1) /* нет аргументов; копируется стандартный ввод */
      filecopy(stdin, stdout);
   else
   while (--argc > 0)
      if ((fp = fopen(*++argv, "r")) == NULL) {
         printf("cat: не могу открыть файл %s\n", *argv);
         return 1;
      } else {
         filecopy(fp, stdout);
         fclose(fp);
      }
   return 0;
}

/* filecopy: копирует файл ifp в файл ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
   int c;
   while ((c = getc(ifp)) != EOF)
      putc(c, ofp);
}
```

Файловые указатели stdin и stdout представляют собой объекты типа FILE*. Это константы, а не переменные, следовательно, им нельзя ничего присваивать.

Функция

int fclose(FILE *fp)

- обратная по отношению к *fopen*; она разрывает связь между файловым указателем и внешним именем (которая раньше была установлена с помощью *fopen*), освобождая тем самым этот указатель для других файлов. Так как в большинстве операционных систем количество одновременно открытых одной программой файлов ограничено, то файловые указатели, если они больше не нужны, лучше освобождать, как это и делается в программе *cat*. Есть еще одна причина применить *fclose* к файлу вывода, - это необходимость "опорожнить" буфер, в котором *putc* накопила предназначенные для вывода данные. При нормальном завершении работы программы для каждого открытого файла *fclose* вызывается автоматически. (Вы можете закрыть *stdin* и *stdout*, если они вам не нужны. Воспользовавшись библиотечной функцией **freopen**, их можно восстановить.)

4.22.1. Ввод-вывод строк

В стандартной библиотеке имеется программа ввода **fgets**, аналогичная программе *getline*, которой мы пользовались в предыдущих главах.

char *fgets(char *line, int maxline, FILE *fp)

Функция *fgets* читает следующую строку ввода (включая и символ новой строки) из файла *fp* в массив символов *line*, причем она может прочитать не более *MAXLINE-1* символов. Переписанная строка дополняется символом '\0'. Обычно *fgets* возвращает *line*, а по исчерпании файла или в случае ошибки - NULL. (Наша *getline* возвращала длину строки, которой мы потом пользовались, и нуль в случае конца файла.)

Функция вывода **fputs** пишет строку (которая может и не заканчиваться символом новой строки) в файл.

int fputs(char *line, FILE *fp)

Эта функция возвращает EOF, если возникла ошибка, и неотрицательное значение в противном случае.

Библиотечные функции **gets** и **puts** подобны функциям *fgets* и *fputs*. Отличаются они тем, что оперируют только стандартными файлами *stdin* и *stdout*, и кроме того, *gets* выбрасывает последний символ '\n', а *puts* его добавляет.

Чтобы показать, что ничего особенного в функциях вроде *fgets* и *fputs* нет, мы приводим их здесь в том виде, в каком они существуют в стандартной библиотеке на нашей системе.

```
/* fgets: получает не более n символов из iop */
char *fgets(char *s, int n, FILE *iop)
{
   register int c;
   register char *cs;

   cs = s;
   while (--n > 0 && (c = getc(iop)) != EOF)
      if ((*cs++ = c) == '\n')
         break;
   *cs= '\0';
   return (c == EOF && cs == s) ? NULL : s;
}

/* fputs: посылает строку s в файл iop */
int fputs(char *s, FILE *iop)
{
   int c;

   while (c = *s++)
      putc(c, iop);
   return ferror(iop) ? EOF : 0;
}
```

Стандарт определяет, что функция *ferror* возвращает в случае ошибки ненулевое значение; *fputs* в случае ошибки возвращает EOF, в противном случае - неотрицательное значение.

С помощью *fgets* легко реализовать нашу функцию *getline*:

```
/* getline: читает строку, возвращает ее длину */
int getline(char *line, int max)
{
   if (fgets(line, max, stdin) == NULL)
```

```
        return 0;
    else
        return strlen(line);
}
```

4.22.2. Дескрипторы файлов

В самом общем случае, прежде чем читать или писать, вы должны проинформировать систему о действиях, которые вы намереваетесь выполнять в отношении файла; эта процедура называется *открытием* файла. Если вы собираетесь писать в файл, то, возможно, его потребуется создать заново или очистить от хранимой информации. Система проверяет ваши права на эти действия (файл существует? вы имеете к нему доступ?) и, если все в порядке, возвращает программе небольшое неотрицательное целое, называемое *дескриптором файла*. Всякий раз, когда осуществляется ввод-вывод, идентификация файла выполняется по его дескриптору, а не по имени. Вся информация об открытом файле хранится и обрабатывается операционной системой; программа пользователя обращается к файлу только через его дескриптор.

4.22.3. Нижний уровень ввода-вывода (read и write)

Ввод-вывод основан на системных вызовах **read** и **write**, к которым Си-программа обращается с помощью функций с именами *read* и *write*.

Для обеих первым аргументом является *дескриптор файла*. Во втором аргументе указывается массив символов вашей программы, куда посылаются или откуда берутся данные. Третий аргумент - это количество пересылаемых байтов.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

Обе функции возвращают число переданных байтов. При чтении количество прочитанных байтов может оказаться меньше числа, указанного в третьем аргументе. Нуль означает конец файла, а -1 сигнализирует о какой-то ошибке. При записи функция возвращает количество записанных байтов, и если это число не совпадает с требуемым, следует считать, что запись не произошла. За один вызов можно прочитать или записать любое число байтов. Обычно это число равно или 1, что означает посимвольную передачу "без буферизации", или чему-нибудь вроде 1024 или 4096, соответствующих размеру физического блока внешнего устройства. Эффективнее обмениваться большим числом байтов, поскольку при этом требуется меньше системных вызовов.

4.22.4. Системные вызовы open, creat,close,unlink

В отличие от стандартных файлов ввода, вывода и ошибок, которые открыты по умолчанию, остальные файлы нужно открывать явно. Для этого есть два системных вызова: **open** и **creat**.

Функция *open* почти совпадает с *fopen*. Разница между ними в том, что первая возвращает не файловый указатель, а дескриптор файла типа *int*. При любой ошибке *open* возвращает -1.

```
include <fcntl.h>

int fd;
int open(char *name, int flags, int mode);

fd = open(name, flags, perms);
```

Как и в fopen, аргумент name - это строка, содержащая имя файла. Второй аргумент, flags, имеет тип int и специфицирует, каким образом должен быть открыт файл. Его основными значениями являются:

O_RDONLY - открыть только на чтение;

O_WRONLY - открыть только на запись;

O_RDWR - открыть и на чтение, и на запись.

O_CREAT - если файл не существует, создать его

O_EXCL - если указан режим O_CREAT, а файл уже существует, то возвратить ошибку открытия файла

O_APPEND - начать запись с конца файла

O_TRUNC - удалить текущее содержимое файла и обеспечить запись с начала файла

O_BINARY - открытие файла в двоичном режиме

O_TEXT - открытие файла в текстовом виде.

Эти константы определены в <fcntl.h>.

По умолчанию файл открывается в двоичном виде.

Третий параметр mode принимает следующие значения:

S_IWRITE - разрешение на запись
S_IREAD - разрешение на чтение
S_IREAD|S_IWRITE - разрешение на чтение и запись.
Эти константы определены в <sys\stat.h>.
Примеры использования:
Open("stock.dat",O_CREAT|O_RDWR,S_IREAD|S_IWRITE) - создать новый файл stock.dat для модификации (чтения и записи)
Open("c:\\dir\\stock.dat",O_RDONLY|O_BINARY) - открыть файл c:\\dir\\stock.dat на чтение в бинарном виде

На количество одновременно открытых в программе файлов имеется ограничение (обычно их число колеблется около 20). Поэтому любая программа, которая намеревается работать с большим количеством файлов, должна быть готова повторно использовать их дескрипторы. Функция **close**(*int fd*) разрывает связь между файловым дескриптором и открытым файлом и освобождает дескриптор для его применения с другим файлом. Она аналогична библиотечной функции *fclose* с тем лишь различием, что никакой очистки буфера не делает. Завершение программы с помощью *exit* или *return* в главной программе закрывает все открытые файлы.

Функция **unlink**(*char *name*) удаляет имя файла из файловой системы.

4.22.5. Произвольный доступ (lseek)

Ввод-вывод обычно бывает последовательным, т. е. каждая новая операция чтения-записи имеет дело с позицией файла, следующей за той, что была в предыдущей операции (чтения-записи). При желании, однако, файл можно читать или производить запись в него в произвольном порядке. Системный вызов **lseek** предоставляет способ передвигаться по файлу, не читая и не записывая данные. Так, функция

long lseek(int fd, long offset, int origin);

в файле с дескриптором *fd* устанавливает текущую позицию, смещая ее на величину *offset* относительно места, задаваемого значением *origin*. Значения параметра origin SEEK_SET, SEEK_CUR или SEEK_END означают, что на величину *offset* отступают соответственно от *начала*, от *текущей позиции* или от *конца* файла. Например, если требуется добавить что-либо в файл, то прежде чем что-либо записывать, необходимо найти конец файла с помощью вызова функции

lseek(fd, 0L, SEEK_END);

Чтобы вернуться назад, в начало файла, надо выполнить

lseek(fd, 0L, SEEK_SET);

Следует обратить внимание на аргумент 0L: вместо 0L можно было бы написать (*long*)0 или, если функция *lseek* должным образом объявлена, просто 0. Благодаря *lseek* с файлами можно работать так, как будто это большие массивы, правда, с замедленным доступом.

Возвращаемое функцией *lseek* значение имеет тип *long* и является новой позицией в файле или, в случае ошибки, равно -1. Функция *fseek* из стандартной библиотеки аналогична *lseek*: от последней она отличается тем, что в случае ошибки возвращает некоторое ненулевое значение, а ее первый аргумент имеет тип *FILE**.

4.22.6. Сравнение файлового ввода-вывода и ввода-вывода системного уровня

Когда следует пользоваться функциями файлового ввода-вывода, а когда функциями ввода-вывода системного уровня? Надо исходить из ответов на два вопроса:

1. что является более естественным для задачи?
2. что более эффективно?

При работе с текстовыми файлами удобнее пользоваться функциями файлового ввода-вывода, в особенности функциями форматированного ввода-вывода. Основным недостатком функций файлового ввода-вывода является то, что они нередко состоят из большого числа команд и резервируют больше памяти для данных, чем функции ввода-вывода системного уровня. Поэтому при использовании функций файлового ввода-вывода программа будет большего размера.

При работе с бинарными данными удобнее пользоваться функциями системного уровня. Этими функциями полезно пользоваться также тогда, когда программа становится слишком большой, либо когда применение функций ввода-вывода верхнего уровня не дает ощутимого преимущества. Функции ввода-вывода нижнего уровня содержат меньше команд и резервируют меньше памяти для данных и могут в зависимости от деталей реализации оказаться более эффективными.