



Инструментарий
Программиста.

Бошкин А.В.

Дубнер П.Н.

РАБОТА В ТУРБО СИ

УНЦ "ТРЭК" СП "ЛАНИТ"
Серия "ИНСТРУМЕНТАРИЙ
ПРОГРАММИСТА"

Бошкин А.В., Дубнер П.Н.

РАБОТА В ТУРБО СИ

Под редакцией:
Генса Г.В., Дубнера П.Н.

Научно-информационная
внедренческая фирма
ЮКИС

Бошкин А.В., Дубнер П.Н. Работа в Турбо Си. - М.:
НИВФ "ЮКИС" при участии - УНЦ "ТРЭК", СП "ЛАНИТ",
1991 г.- 183 с. Цена договорная.

ISBN-5-86030-013-1

Эта книга - руководство к действию для пользователей Турбо Си, желающих повысить эффективность своего труда. В книге содержится описание среды Турбо Си как инструмента создания программ, приводятся советы технологического порядка - как создавать многомодульные программы, как работать с библиотеками объектных программ, как избавиться от предупреждений, выдаваемых компилятором и т.д. Детально рассматриваются версии языка Си, встроенного в операционное окружение Турбо Си.

ISBN-5-86030-013-1

© СП "Ланит"

ПРЕДИСЛОВИЕ ИЗДАТЕЛЯ

В процессе деятельности профессиональному программисту приходится изобретать подходящие для его конкретной задачи инструменты - от средств самого высокого уровня (например, обеспечивающих "естественно-языковой" интерфейс) до утилит, позволяющих ему подменять некоторые или почти все функции операционной системы. Разработка подобного инструментария всегда сопряжена по крайней мере с двумя опасностями.

Прежде всего, часто нам приходится "изобретать велосипед", причем из-за отсутствия времени и/или соответствующего опыта, велосипед получается довольно плохого качества. Во-вторых, затраты на инструментарий часто оказываются неожиданно большими и приводят к опозданию в выполнении основного проекта. Решением возникающих здесь проблем был бы прямой контакт с теми, кто решает аналогичные проблемы. Это, однако, не всегда осуществимо.

Мы начинаем публиковать серию книжек под общим названием "Инструментарий программиста", в которой профессиональные программисты высокого класса будут рассказывать о своих подходах к решению задач, почти неизбежно возникающих при работе над сколько-нибудь нетривиальным программным проектом.

Первые выпуски серии предназначены для тех, кто программирует на языке Си. Си - один из самых критикуемых и, вместе с тем, один из самых сейчас распространенных языков программирования. Его адепты похваляются его скороговоркой и гибкостью, его критики указывают на его темные места и предлагают работать на Паскале или Модуле. Не споря ни с кем, скажем лишь, что профессиональному программисту почти неизбежно придется в своей деятельности написать на Си по крайней мере одну программу. Даже тем, кто программирует на Лиспе или Прологе, приходится встраивать в свои программы куски текста плохо или вовсе не выраженные на избранном языке; почти во всех Лисп- и Пролог- и т.п. системах этот текст может быть написан либо на языке ассемблера, либо на Си, но не на Паскале или другом языке программирования, удовлетворяющем вкусам пуристов. С этой точки зрения Си можно рассматривать как язык ассемблера высокого уровня, что и является причиной того, что мы начинаем с него.

Турбо Си - очень мощный и широко распространенный инструмент программирования. Наличие интегрированной среды разработки программ привлекло на его сторону множество поклонников, которые в других обстоятельствах, возможно, выбрали бы Микрософт Си, Латтис Си или другую реализацию Си. Несмотря на появление на рынке в последнее время систем, также обладающих интегрированной средой (таковы, например, Квик Си и Турбо Си++), можно с определенной степенью уверенности предполагать, что массовое использование Турбо Си будет продолжаться еще довольно долго. Так, Квик Си не предлагает ничего принципиально нового (смена же привычной среды на другую требует каких-то очень серьезных выгод),

среда же Турбо Си++ пока что очень сложна в использовании и обладает существенно более медленным компилятором.

По поводу проблем, с которыми сталкиваются пользователи Турбо Си, можно сказать уверенно: главная из них - отсутствие документации. причем имеется в виду документация в широком смысле: руководства, справочники, учебники, сборники советов - как по языку, так и по самой системе. Поэтому Вашему вниманию предлагается книга, в которой рассматриваются все составные части системы программирования Турбо Си. Упор делается на малоиспользуемые возможности, в то же время без внимания не оставлена ни одна значительная деталь. Значительное внимание уделено технологии применения языка Си в Турбо-системе. Там, где это уместно, приведены достаточно развернутые примеры с подробными объяснениями.

Уровень подготовки читателя предполагается не слишком высоким, но и не нулевым (на уровне "библии по Си" - книги Кернигана Б. и Ритчи Д.). Так, информация, которая могла быть интересной и понятной лишь для специалистов высокого полета, в книге отсутствует - вместо нее имеются краткие комментарии и отсылки к руководству.

Мы надеемся, что данная книга окажется полезной для широкого круга программистов и реализация наших планов по дальнейшему расширению серии будет во многом зависеть от того, как читатели примут первую книжку серии "Инструментарий программиста".

Ваши отзывы и пожелания направляйте по адресу: 107078, Москва, Новая Басманная ул., д.16 а, УНЦ "ТРЭК".

Генеральный директор СП "ЛАНИТ" Г.В. Генс

ПРЕДИСЛОВИЕ

Скажите, что вы подумаете о шофере, который на вопрос "А зачем эта кнопка на приборной доске вашей машины?" ответит: "Не знаю, никогда ей не пользовался"? Вряд ли вы после такого ответа будете высокого мнения об уровне его водительского мастерства. А теперь - несколько похожих вопросов из другого рода деятельности:

- А что будет, если в редакторе Турбо Си нажать Ctrl-O I?
- Зачем нужна программа BGIOBJ (одна из утилит Турбо Си)?
- Что делает команда "Find Function" из меню Debug?

Подобных вопросов о Турбо Си можно придумать множество. А много ли программистов - конечно, из работающих в этой системе, - способных без раздумий ответить на каждый из них? Как говорится, подавляющее меньшинство. Конечно же, такое рассуждение справедливо не только для Турбо Си, но и для любой другой сколько-нибудь сложной системы. Это говорит о том, что овладение всеми имеющимися возможностями ежедневно используемого инструмента еще не стало неотъемлемым элементом культуры труда программиста. К сожалению, это отличает нашу профессию от множества других, зачастую не менее сложных (подумаем, например, о пилотах или хирургах).

Цель данной книжки - показать читателю богатство Турбо Си как инструмента создания программ, осветить все самые дальние закоулки его меню и редко используемые, чаще по незнанию, команды, показать, как использовать утилиты (это ведь действительно "пользухи", часто облегчающие жизнь). Мы также постараемся дать кое-какие советы технологического порядка - как аккуратно управляться с многомодульными программами, как избавиться от предупреждений, выдаваемых компилятором, когда и как использовать библиотеки объектных программ и так далее.

Эта книжка, конечно, не превратит читателя в кудесника, выжимающего из Турбо Си все, на что только способна эта система, - тут мало будет любого количества печатного материала, если нет повседневного опыта работы в Турбо Си или, например, желания совершенствоваться. Наша книжка это, так сказать, руководство к действию для пользователей Турбо Си, желающих повысить эффективность своего труда. Она призвана заполнить пробелы в знании системы, особенно у тех, кто не имеет возможности прочесть фирменную документацию в силу ее отсутствия или слабого знания языка. Возможно, что-то интересное в ней найдут для себя даже те, кто уже давно освоил все или почти все возможности системы и считается среди коллег экспертом по работе в Турбо Си. Новичкам книга, по-видимому, будет мало полезна сейчас и пригодится лишь после овладения языком Си и приобретения известных навыков в использовании системы.

Книгу стоит читать подряд, не стараясь сразу запомнить все, что в ней написано: цель прочтения будет достигнута, если вы проникнетесь

“духом” системы и почувствуете, что вопросов о ней, способных поставить вас в тупик, стало меньше.

Использовать книгу в качестве справочника большого смысла не имеет: лучший справочник - “Руководство пользователя”; система помощи Турбо Си в этом смысле тоже хороша.

В общем, перед вами - не учебник и не справочник, а, так сказать, неформальный курс повышения квалификации пользователя Турбо Си. По этой причине, в частности, изложение не сковано рамками строгости и единства терминологии: так, вполне возможно использование еще не описанных понятий, отдельные элементы меню называются в разных местах “позициями”, “переключателями”, “опциями”, термины “трансляция” и “компиляция” используются как полные синонимы, и так далее. Авторам кажется, что в книге такого рода, как эта, излишний академизм неуместен.

Предполагается, что читатель умсет программировать на языке Си и имеет доступ к компьютеру, оснащенному системой Турбо Си версии 2.0 (большая часть материала в равной степени относится и к версии 1.5, в которой беднее меню, отсутствует встроенный отладчик и часть описываемых утилит).

Книга включает две части. Первая часть “Операционное окружение Турбо Си” содержит:

Глава 1 - “Основные возможности” - представляет собой беглый обзор команд и меню интегрированной системы.

Глава 2 - “Редактор” - знакомит с основными приемами работы с встроенным редактором Турбо Си, подробно рассматриваются не слишком популярные команды редактора.

Глава 3 - “Компилятор” - содержит обзор опций компилятора интегрированной системы с рекомендациями по их использованию. Объясняется смысл предупреждающих сообщений компилятора, обсуждаются способы их устранения.

Глава 4 - “Сборка выполняемой программы” - посвящена рассмотрению трудностей, которые могут возникнуть на этом этапе. Приводятся рекомендации по использованию опций линкера.

Глава 5 - “Управление проектом” - дает понятие о том, как аккуратно строить многомодульные программы, как создавать файл проекта и как пользоваться командой интегрированной среды Make.

Глава 6 - “Отладка в интегрированной среде” - описывает возможности встроенного отладчика Турбо Си и действия, необходимые для подготовки программы к отладке.

Глава 7 - “Работа из командной строки” - это введение в использование средств Турбо Си, позволяющих работать с компилятором и редактором связей в среде DOS. Также рассматривается утилита make.

Глава 8 - “Утилиты” - показывает, как пользоваться различными утилитами из числа имеющихся на дистрибутивных дисках Турбо Си.

Вторая часть книги “Справочное руководство по Турбо Си” посвящена той версии языка Си, которая встроена в операционное окружение и поддерживается им. Некоторыми тонкими деталями она расширяет стандарт (так называемый ANSI Си), поэтому мы сочли необходимым всюду указывать, чем именно данная версия отличается от стандарта языка.

Дубинер П.Н., Бошкин А.В.

ЧАСТЬ 1. ОПЕРАЦИОННОЕ ОКРУЖЕНИЕ ТУРБО СИ

1. ОСНОВНЫЕ ВОЗМОЖНОСТИ

Начиная разговор об интегрированной среде разработки Турбо Си (ниже будем называть ее просто средой), мы должны, казалось бы, пуститься в подробное описание всего, что пользователь среды видит на экране: окно такое, окно сякое, меню... Текст получится столь же большой, сколь и утомительный: с этим согласится всякий, кто читал соответствующую главу Руководства пользователя Турбо Си. Мы твердо верим, что любому, кто хотя бы немного поработал с Турбо Си (именно на этот круг читателей рассчитана книга), неинтересно читать о том, где на экране искать основное меню среды, как вызывать систему помощи, наконец, как выходить обратно в ДОС. С другой стороны, вещи такого рода должны быть хотя бы упомянуты - например, для того, чтобы не бояться упреков в неполноте материала.

Так и решено: в главе, конечно, будет рассказано об окнах среды, об основном меню, о строке-подсказке и системе помощи, но вещи общеизвестные будут лишь упоминаться. Основное внимание мы уделим сведениям, на наш взгляд, полезным, однако известным не каждому.

1.1. ОКНА

Каждому пользователю Турбо Си известно, что кроме окна редактора (оно так и называется "Edit") в интегрированной среде Турбо Си (напомним, ниже она - просто "среда") имеются еще окно, куда выводятся сообщения компилятора и редактора связей ("Message"), и окно "Watch", куда отладчик среды выводит значения трассируемых переменных.

Одновременно на экране можно видеть одно или два из трех упомянутых окон. Переключение между этими двумя режимами производится клавишей F5 (это, кстати говоря, очень популярная в системах, работающих с окнами, операция "Zoom", правильный перевод выглядит слишком уж длинно: "расширить активное окно до полного экрана", жаргонный перевод "зумнуть" гораздо короче - выбирайте сами).

Сначала поговорим о режиме, когда экран разделен на два окна. При этом верхним всегда будет окно редактора, а нижним - "Message" или "Watch". Решение ограничиться только двумя окнами, одновременно можно понять. Действительно, во время отладки редко бывают интересны сообщения, выданные при последней компиляции и редакции связей, и наоборот.

в момент работы с компилятором значения трассируемых переменных, как правило, не определены. Ясно, что держать на экране постоянно все три окна - безответственная трата его дефицитных строк.

Размер каждого из окон не постоянен. Любое из окон в нижней части экрана растет по мере накопления в нем информации (уменьшая тем самым размер окна редактора), но не более, чем до фиксированного размера, установленного ранее при помощи утилиты TCINST (о ней, как и о других утилитах Турбо Си, речь пойдет в соответствующей главе).

В каждый момент времени одно из двух окон является текущим. Какое именно, определить проще простого: если текущее окно - окно редактора, в нем есть курсор, если "Message" или "Watch", то на манер меню подсвечена одна из строк этого окна. Мало того, верхняя кромка текущего окна всегда нарисована двойной линией.

Эффект уже упомянутой команды "Zoom" (клавиша F5) состоит в том, что текущее окно начинает занимать весь экран целиком (конечно, кроме верхних двух строк - меню и статуса - и нижней - подсказки), для окна редактора эффект усиливается еще и удалением рамки. Переход обратно в двухоконный режим выполняется той же самой клавишей F5.

Смена текущего окна происходит либо автоматически (например, после завершения компиляции, если выданы какие-то сообщения), либо по велению пользователя. Команда перехода из окна редактора в "нижнее" окно ("Message" или "Watch") и обратно - F6. Из окна "Message" в окно редактора - F6 или Enter. Для перехода из окна "Message" сразу в окно "Watch" (и обратно) нужно нажать Alt-F6. Обратите внимание: та же клавиша, нажатая в момент, когда текущее окно - окно редактора - выполняет совсем другую функцию: для обработки загружается файл, который редактировался последним перед текущим (подробности - ниже в этой главе).

Пару слов о содержимом окна "Message" (об устройстве окна "Watch" рассказывается в главе, посвященной отладчику, отдельная же глава знакомит с редактором, "живущим" в своем окне).

Как уже говорилось в самом начале главы, в окно "Message" выводятся сообщения компилятора. Сообщения эти бывают трех сортов: просто информация (например, "компилирую файл такой-то"), предупреждения (переход к шагу редактирования связей возможен, но есть смысл задуматься над происхождением сообщения) и ошибки (придется внести исправления в программу и повторить компиляцию). Все сообщения накапливаются в списке в порядке поступления и после окончания трансляции представляются пользователю (окно "Messages" при этом автоматически становится текущим).

Интересное свойство этого списка в том, что он устроен как меню, и выбор из этого меню (клавишей Enter) какого-либо сообщения приводит к переходу в окно редактора, причем курсор устанавливается в точку программы, ответственную за сообщение (точнее, в строку, при компиляции которой оно порождено). Более того, если окно "Message" находится на экране одновременно с окном редактора, то при передвижении по "меню" сообщений в окне редактора подсвечивается соответствующая за текущее сообщение строка программы, причем при необходимости может быть загружен и другой файл (например, включенный в текст директивой # include). Последнюю возможность можно отключить или ограничить (не производит-

ся загрузка другого файла) установкой в соответствующее положение переключателя "Message Tracking" меню "Options/Environment".

Наконец, последнее, что можно еще сказать об устройстве окон в среде Турбо Си: имеется возможность посмотреть на так называемый "пользовательский экран". В любой момент, когда среда ожидает команду, пользователь может нажать Alt-F5 и полюбоваться на экран с остатками сообщений ДОСа и/или последними по времени строками, выведенными его программой (через stdout или каким-либо другим образом) на экран. Возврат из просмотра пользовательского экрана к экрану среды происходит после нажатия любой клавиши.

1.2. МЕНЮ

Самая верхняя строка экрана среды Турбо Си отведена под главное меню. В нем сосредоточено большинство доступных пользователю команд среды.

Входят в главное меню нажатием клавиши F10, после чего клавишами "стрелка влево" и "стрелка вправо" выбирают нужную позицию, затем нажатием Enter раскрывают меню (вертикальное) следующего уровня, соответствующее выбранной позиции. Впрочем, так делают в основном те, кому не дорого время, проводимое за компьютером, а также новички и скушающие люди в состоянии "чем бы таким заняться". Моментальный вход в меню второго уровня из любого места среды, настойчиво рекомендуемый остальным пользователям, - нажатие клавиши Alt одновременно с первой буквой имени одной из позиций главного меню F, R, C, P, O, D, B. Тот же эффект имеет последовательное нажатие F10 и одной из перечисленных букв. Заметим, что (в отличие от остальных) позиция "Edit" меню второго уровня не имеет, и нажатие Alt- E (или F10, затем E) всегда приводит к переходу прямо в редактор.

В вертикальных меню второго и следующих уровней передвигаются при помощи стрелок вверх и вниз, а позицию в меню выбирают клавишей Enter. Здесь тоже имеется более быстрый путь - как обычно в системах меню такого рода, можно моментально выбрать нужную позицию (команду или меню следующего уровня) нажатием первой буквы ее названия. Например, если вы вошли в меню "File", оказались при этом на строчке "Load" и хотите посмотреть содержимое директория (позиция "Directory"), совсем не обязательно пять раз нажимать "стрелку вниз" или четыре - "стрелку вверх", а затем Enter. Стукните по кнопке D и получите желаемое гораздо быстрее.

Стоит оговориться, что выбор из любого меню нажатием первой буквы названия позиции может не работать в случае, если клавиатура находится в режиме русских букв (или в каком-то другом, отличном от стандартного).

Часть команд имеет также альтернативную форму выбора - так называемые "горячие клавиши" (уж извините за рабский перевод термина hot keys на родной язык). Нажатием горячей клавиши можно вызвать команду из любого места среды, не прибегая к использованию меню. Пример: экономным эквивалентом команды "Quit" из меню "File" является нажатие Alt-X.

Во всех меню, начиная с третьего уровня, нажатие Esc приводит к выходу в меню предыдущего уровня, со второго уровня Esc переводит прямо

в текущее окно экрана. Повторим, ибо это большое удобство: сразу сквозь все уровни меню можно зыйти в редактор нажатием Alt-E.

Посмотрим на команды, скрывающиеся под главным меню. Большая их часть рассматривается в главах, посвященных соответствующим подсистемам (все команды позиций "Run", "Compile", "Project", "Debug" и "Break/watch", а также некоторые команды позиции "Options"). Остаются "File" и часть команд позиции "Options". О них здесь и пойдет речь.

Команды меню второго уровня "File" ведают, так сказать, организационными вопросами - сменой файлов, общением с ДОС и завершением сеанса. Рассмотрим все его позиции; для команд, имеющих горячие клавиши, обозначение их приводится в скобках после названия команды.

"Load" (F3). *Загрузить в окно редактора новый файл.* Команда открывает маленькое окошко, в котором предлагается ввести имя загружаемого файла. В качестве заготовки для имени в окне находится текст, который вы набрали здесь при прошлом использовании команды "Load", или, если еще ей не пользовались, строка "*.C". Часто бывает, что эта заготовка помогает заметно сэкономить время на набор. Так, если вы попытались загрузить файл с каким-нибудь длинным именем и ошиблись, нажмите F3 еще раз и отредактируйте имя файла.

Чтобы приступить к редактированию, достаточно первой нажать одну из клавиш: Home, PgUp, End, PgDn, Ins, Del, стрелка вправо, стрелка влево. Фон, на котором выведена заготовка, изменится, и можно заниматься обычным редактированием внутри строки. Нажатие любых "буквенных" клавиш очищает поле ввода имени файла, предоставляя пользователю вводить все имя от начала до конца.

В имени файла можно пользоваться обычными для ДОС символами wildcards (здесь по аналогии с "горячими клавишами" можно было бы привести перевод "дикие карты", но на это духу не хватает) - "*" и "?", а также указывать маршрут. Если имя файла однозначно определяет файл (то есть в имени не использованы wildcards) файл загружается для редактирования, причем если такой файл не найден, он считается пустым. В противном случае пользователю предлагается выбрать имя нужного файла из всех подходящих под заданный им образец имен файлов указанного директория. Имена эти выводятся в отдельное окно в виде меню, и там же есть имена всех поддиректориев, а также, возможно, обычное наименование верхнего директория, представленное в виде ".. \".

Выбрав в этом меню файл, вы получите его в окне редактора. Выбрав один из директориев, сможете выбирать файлы, записанные в нем (но опять же из тех, имена которых подходят под образец), или продолжить путешествие по файловой системе.

Еще одно замечание о вводе имен файлов. Если пользователь задает имя файла без расширения, автоматически предполагается, что его интересуют файлы с расширением ".C" (часто так оно и бывает). Если имя файла не имеет расширения, то после имени следует поставить точку.

Если файл, который вы редактировали ранее, не сохранен на диске, система спросит, сохранять ли его перед загрузкой нового файла. В ответ нажимается клавиша "Y" (да, сохранять) или "N" (пусть изменения пропадают).

Нажатие клавиши Esc в любой момент описанных манипуляций вернет вас в точку, из которой вызвана команда "Load", без каких-либо изменений в тексте файла.

"Pick" (Alt-F3). Загрузить в окно редактора один из недавно редактированных файлов. Великолепное изобретение, сэкономившее по всему миру, видимо, не один человеко-год программистского времени. Команда выводит меню, в котором перечислено до девяти файлов. Первый из них - файл, редактируемый в данный момент, остальные - те, которые редактировались ранее, причем изначально меню указывает на файл, редактировавшийся непосредственно перед текущим. Выбрав файл из этого меню, вы получаете его на экране, как по команде "Load". На случай, если интересующего вас файла в предъявленном списке нет, десятая позиция меню - обычная команда "Load", впрочем, F3 здесь тоже работает.

Команды "Load" и "Pick" имеют еще одно применение, кроме загрузки новых файлов. Предположим, вы внесли в файл какие-то изменения и... передумали, не успев еще эти изменения записать на диск. Посредством одной из этих команд загрузите тот же самый файл и в ответ на законный вопрос "сохранять ли изменения" нажмите "N". Файл свой вы получите в первоначальной красе. Особенно удобна для этого команда "Pick" - вместе с "N" нужно всего 4 заранее известных нажатия на клавиши (каких именно - упражнение для любознательных; для читателей, склонных к поиску совершенства во всем, сообщим, что есть способ из трех нажатий).

"New". Начать редактирование нового (пустого) файла. Новому файлу дается имя "NONAME.C". В дальнейшем, по всей видимости, придется этот файл переименовать (при сохранении NONAME.C среда сама предлагает это сделать). Альтернативный способ заведения нового файла - при помощи команды "Load" (ей можно задать имя несуществующего файла, тогда он будет считаться новым и пустым). Здесь уже переименование не потребуется, но имя нового файла нужно придумать заранее.

"Save" (F2). Запись редактируемого файла на диск. Файл записывается в тот директорию, из которого был считан для редактирования. Новый файл помещается в директорию, который был текущим в период создания файла. Если в этом директории уже есть файл с таким именем, среда спрашивает, стирать ли старый файл (Y - стирать, N или Esc - вернуться в редактор без каких-либо действий).

"Write to". Переименовать редактируемый файл и записать на диск под новым именем. У пользователя запрашивается новое имя файла. Если файл с таким именем на диске уже есть, как обычно, уточняется, можно ли его затирать. При желании можно записать переименованный файл в другой директорию, тогда придется набирать соответствующий маршрут. Набор маршрута можно облегчить, если при задании нового имени файла использовать wildcards. В этом случае пользователю будет предложен список подходящих под образец имен существующих файлов, а также предоставлена возможность перейти в другие директории. Файлы будут перечислены в меню, таком же, как у команд загрузки. Выбор файла в этом меню будет означать запись редактируемого файла на место выбранного (по-прежнему с подтверждением).

"Directory". Вывести содержимое директории с возможностью выбора файла. Эта команда практически дублирует команду "Load".

Остановимся на минуту и рассмотрим подробнее тот список файлов, который предъявляют для выбора команды "Load", "Write to" и "Directory". Уже было сказано, что список устроен, как меню: передвигаться по нему можно с помощью клавиш Home, End, PgUp, PgDn и стрелок, выбирать файл или директорию нажатием Enter. Кроме этого, есть и еще довольно

удобная возможность: если нажать клавишу с буквой, подсветка меню передвинется на первый из файлов, чье имя начинается с этой буквы (файлы в меню расположены в алфавитном порядке). Список включает и поддиректории (они сгруппированы в конце списка, упорядочены тоже по алфавиту), а также специальный файл "..\", означающий родительский директорию по отношению к показываемому. Поддиректорий также можно выбирать по первой букве имени, нажимая Shift и эту букву.

Вверху окна написан образец (маска), по которому отобраны файлы для выбора. Если вам вздумалось изменить эту маску, совсем не обязательно повторять вызов команды - нажимите F4 и сможете, не выходя из меню имен файлов, задать новый образец.

Двинемся дальше по меню "File".

"Change dir". *Смена текущего директория*. Предлагается ввести название директория, который отныне станет текущим. Возможность, конечно, иногда полезная, но пользоваться ей нужно с осторожностью, так как практика показывает, что часто следствием смены директории "на ходу" является путаница в расположении файлов.

"OS shell". *Временный выход в среду ДОС*. Вы имеете возможность вызвать любую команду ДОС или программу, а затем вернуться обратно в среду Турбо Си. Здесь также следует с осторожностью применять команду смены директория. Заметим также, что при работе "из-под" Турбо Си свободной остается гораздо меньшая часть оперативной памяти, так что некоторые программы могут не пойти по причине нехватки памяти. Для возврата в Турбо Си, как обычно, используется команда EXIT. Все, что было на экране в момент возврата, остается и доступно обозрению после нажатия Alt-F5.

"Quit" (Alt-X). *Покинуть среду*. Команда, с которой нужно начинать знакомство с любой системой. Единственное, что можно здесь сказать: если вы пытаетесь выйти из среды, не записав на диск изменения, внесенные в последний из редактируемых файлов, Вам напомнят об этом (как обычно, Y - записать файл и выйти, N - выйти без записи, Esc - остаться в среде).

Перейдем к знакомству с меню "Options". Меню второго уровня "Options/Compiler" и "Options/Linker" подробно рассматриваются в следующих главах, здесь же описаны остальные 5 позиций этого меню.

"Environment". *Установка параметров среды*. Здесь содержатся команды, позволяющие устанавливать режимы работы среды, а именно:

"Message Tracking". *Отслеживание сообщений*. Как уже говорилось, при передвижении по списку сообщений компилятора в окне "Messages" возможно одновременное передвижение подсветки в окне редактора, обозначающей строку текста программы, к которой относится текущее сообщение. Есть три варианта такого отслеживания, каждому из них соответствует значение переключателя "Message Tracking".

"All files" (*все файлы*): если сообщение относится к строке другого файла (не того, который находится в окне редактора), нужный файл будет автоматически загружаться в окно редактора.

"Current file" (*текущий файл*): отслеживаются только сообщения, относящиеся к текущему файлу. Если в окне "Messages" подсветка установлена на сообщение, относящееся к другому файлу, подсветка в окне редактора пропадает.

"Off": сообщения не отслеживаются вовсе. Тем не менее, нажатием **Enter**, как и в остальных режимах отслеживания, можно перейти в окно редактора к соответствующей текущему сообщению строке.

Для того, чтобы изменить значение переключателя, нужно установить подсветку меню **"Options/Environment"** в позицию **"Message Tracking"** и нажимать **Enter** до тех пор, пока переключатель не примет требуемое значение.

"Keep messages". *Сохранение сообщений*. Переключатель на два положения, обозначаемыми словами **"Yes"** и **"No"**, управляет порядком очищения окна **"Messages"** от выданных ранее сообщений. Если значение переключателя - **"Yes"**, хранятся все сообщения, и лишь при повторной компиляции файла из списка удаляются те сообщения, что были выданы при предыдущей компиляции. Если значение - **"No"**, список сообщений очищается перед каждой компиляцией.

"Config auto save". *Автоматическое сохранение конфигурации*. Установка всех переключателей из меню **"Options"** и всех его подменю можно сохранить на диске, в так называемом файле конфигурации. При входе в среду изначально устанавливаются такие их значения, какие записаны в файле конфигурации. Сохранение конфигурации делается вручную (см. ниже - команда **"Options/ Save options"**) и/или автоматически. Если переключатель **"Config auto save"** установлен в **"On"**, среда сама будет сохранять конфигурацию в некотором файле (о том, в каком именно, речь пойдет ниже) перед выходом из среды, перед запуском программы на выполнение и перед вызовом ДОС (команда **"Files /OS shell"**). Если переключатель установлен в **"Off"**, всю ответственность за сохранение вашей конфигурации среды вам придется взять на себя.

"Edit auto save". *Автоматическое сохранение редактируемого файла*. Переключатель похож по действию на предыдущий. Его значение влияет на сохранение редактируемого файла (конечно, если в него внесены изменения) перед запуском программы и перед выходом в ДОС командой **"File/OS shell"**: **"On"** - сохранение производится автоматически, **"Off"** - нет. Отметим, что решение о том, сохранять ли изменения при выходе из среды, остается за пользователем. Разумно держать этот переключатель в положении **"On"**, так как в горячке отладки запросто можно потерять исправления, только что внесенные в программу, и лишь потому, что при очередном запуске вдруг "повисла" машина (как чаще всего бывает, по вине самой же программы).

"Backup files". *Файлы аварийного копирования*. При записи файла на диск среда способна создать "аварийную" копию вашего файла в том состоянии, в каком он был перед началом последнего редактирования. С одной стороны, эта копия иногда бывает чрезвычайно полезна (предположим, файл погиб, - у вас есть возможность восстановить его содержимое за вычетом последних модификаций), с другой, она занимает место (иногда немалое) на диске, что не всегда приемлемо, особенно в наших условиях коммунальных винчестеров. Выбирайте: **"On"** - перед каждой командой **"Save"** среда запомнит содержимое файла под тем же именем, но с расширением **".BAK"**. Если выберете **"Off"**, то сэкономите место и время, но в один прекрасный день можете заплакать горючими слезами.

"Tab size". *Величины табуляции*. Параметр управляет числом пробелов, в которое отображается на экране каждый символ табуляции в редактируемом файле. Обычно число это равно 8, и не видно причин когда-либо

его менять. Интересно, что эта возможность отсутствовала в Турбо Си версии 1.5.

"Zoomed windows". Переключение окон. Тоже не слишком полезная возможность, поскольку этот переключатель (однооконный-двухоконный режим, **"On"**-**"Off"**) дублирует действие, выполняемое гораздо более удобным способом - нажатием клавиши F5

"Screen size". Размер экрана. Если ваш дисплей и адаптер способны работать в 43-строчном или 50-строчном режиме, вы можете найти полезным или приятным иметь больший обзор редактируемого файла. Если нет - утештесь тем, что при работе в обычном 25-строчном режиме меньше устают глаза.

Следующее меню в **"Options"** - меню директориев. Коротко его назначение можно описать фразой "что где искать и куда записывать". Итак, меню **"Directories"**. Для задания требуемого директория следует подвести подсветку к нужной позиции меню и нажать Enter. Среда выведет окошко для ввода нового значения, при этом старое будет написано в этом окне и его можно отредактировать. Для этого надо нажать первым делом одну из клавиш: **Home**, **PgUp**, **End**, **PgDn**, **Ins**, **Del** или любую клавишу со стрелкой. Если допускается задание сразу нескольких директориев, их имена следует разделять точкой с запятой.

"Include directories". Перечисленные директории (в том порядке, в каком они записаны) просматривает компилятор в поисках файлов, включаемых в программу инструкцией **#include**. Для **include**-файлов, чьи имена заключены в двойные кавычки, кроме указанных, причем первым, просматривается текущий директорий. Вы можете уничтожить разницу в обработке **include**-файлов, заключенных в двойные кавычки и в угловые скобки, указав в списке директориев поиска первым текущий директорий (как обычно, он обозначается точкой).

"Library directories". Библиотечные директории. В указанных здесь директориях редактор внешних связей ищет библиотеки объектных программ. Первым просматривается текущий директорий, даже если он не указан.

"Output directory". Выходной директорий. Директорий, в который записываются создаваемые компилятором и редактором связей объектные файлы (расширения **".OBJ"** и **".EXE"**). Здесь же ищутся **OBJ**-файлы для создания выполняемой программы. Если не задано никакого директория, его роль выполняет текущий.

"Turbo C directory". Директорий Турбо Си. Для ответов на вопросы, возникающие у пользователя во время работы в Турбо Си, среда имеет развитую систему помощи (**help**). Все тексты подсказок хранятся в файле **TCHelp.Tch**. Среда должна знать имя директория, в котором записан этот файл, чтобы иметь возможность найти его при работе в других директориях. Оно и задается в этой позиции меню. В этом же директории среда (во время входа в нее) ищет файл конфигурации, если такового нет в директории, откуда вызвана сама программа среды **TC.EXE**.

Наконец, при очередном входе в среду информация о файле, который редактировался перед последним выходом из нее, позволяющая войти "в то же место", на котором была прервана работа, а равно и список недавно редактировавшихся файлов (для команды **"Pick"**) также хранятся отдельно. Если файл с этой информацией не найден в текущем на момент вызова среды директории, то он ищется в директории, указанном здесь. Следует в

заключение указать, что при использовании версии ДОС 3.0 (и выше) особое внимание на этот параметр обращать не обязательно - среда также ищет все перечисленные файлы в том директории, где расположена сама программа среды (ТС.EXE), и, как правило, находит...

"Pick file name". *Имя пик-файла.* Как говорилось в предыдущем абзаце, информация о файле, редактировавшемся в момент выхода из среды (такая, как строка и позиция в строке, на которой был курсор, точное расположение файла и т.п.), а также о файлах, редактировавшихся ранее, записывается в особый файл, называемый пикфайлом (pick-file). Если не предпринимать никаких действий, связанных с заданием пик-файла, он будет иметь имя TSPICK.TSP и расположен в том же директории, что и программа среды (ТС.EXE).

Если же вы желаете (вполне обоснованно) для каждого директория, из которого вызывается Турбо Си, иметь пик-файл с историей работы, ведущейся именно в этом директории, вам нужно, вызвав Турбо Си, выбрать в меню эту позицию и ввести имя пик-файла (собственно, вводить, скорее всего, ничего не потребуется, так как вполне подходящее для пик-файла имя TSPICK.TSP выводится в окне-приглашении автоматически, и достаточно нажать Enter). Чисто умозрительно ясно, что с пик-файлами можно производить и более сложные действия, чем заведение одного файла в каждом рабочем директории, например, иметь по пик-файлу на несколько "стандартных рабочих ситуаций", но выгода от такой суеты представляется сомнительной.

Наконец, позиция меню "Options/Directories" позицией меню лишь прикидывается - вам никогда не удастся установить на нее подсветку. В позиции "Current pick file" всего лишь записано имя текущего пик-файла.

"Arguments". *Аргументы.* Средство задания параметров программы, которые будут переданы в качестве аргументов функции main как пришедшие из командной строки. Набираются они точно в том же виде, как при вызове программы из ДОС (но без имени вызываемой программы). Стоит здесь отметить, что, в отличие от настоящего вызова из командной строки, в среде невозможно перенаправление файлов стандартного ввода и стандартного вывода (stdin и stdout), так что stdin - всегда клавиатура, а stdout - всегда экран. В случае необходимости использовать перенаправление придется выйти из среды или вызвать ДОС командой "File/OS shell" (в последнем случае следует считаться с возможной нехваткой оперативной памяти для выполнения программы).

"Save options". *Сохранение опций.* Установив нужные вам опции компилятора, загрузчика, отладчика, директории для поисков и т. д., то есть создав конфигурацию среды, вы можете сохранить эту конфигурацию в файле для последующего использования (гораздо удобнее установить все опции разом, считав конфигурацию описанной ниже командой "Restore options", чем каждый раз устанавливать их вручную, обходя немалое количество меню). Установив в меню "Options" подсветку на позицию "Save options" и нажав Enter, вы получите приглашение на ввод имени файла конфигурации. Задав имя файла, вы сохраните в нем все опции компилятора (задаваемые в меню "Options /Compiler"), редактора связей (меню "Options/Linker"), встроенного отладчика ("Debug"), параметры проекта ("Project"), настройку самой среды ("Options/Environment") и директории для поиска файлов ("Options/ Directories").

Разумно связывать это имя со смыслом опций, например. "TCDEBUG.TC" - опции для компиляции и работы с использованием отладчика, "TCCOMBAT.TC" - опции "боевой" компиляции с отключенными проверками и отсутствием отладочной информации и так далее. Для тех, кому лень выдумывать осмысленные имена файлов конфигурации, среда предлагает свое название - "TCCONFIG.TC". Можно также ввести только часть названия, так называемую маску имени файла, например, "*.TC", тогда среда предложит выбрать имя файла для записи конфигурации из списка файлов, имена которых соответствуют маске (при этом конфигурацию можно записать даже в директории, не являющемся текущим), при выборе файла из этого списка старое содержимое файла пропадает. Если же в ответ на приглашение ввести пустую строку, конфигурация будет записана в файл "TCCONFIG.TC" директория Турбо Си, то есть директория, указанного в параметре "Options/Directories/Turbo C directory".

"Restore options". *Восстановление опций.* Действие, обратное предыдущему, - загрузить конфигурацию из файла, созданного командой "Options/ Save options". Предлагается ввести имя файла конфигурации, можно задать маску, тогда будет предложен список файлов. Все опции и параметры, записываемые в файле конфигурации, получают свои значения из нового файла.

1.3. СИСТЕМА ПОМОЩИ

Помимо окон и основного меню на экране среды Турбо Си постоянно присутствует еще и строка подсказок - самая нижняя на экране. Она содержит названия наиболее нужных (конечно, по мнению разработчиков среды Турбо Си, которое, как говорится, не обязательно совпадает с официальной точкой зрения Конгресса) команд, причем с каждым из трех основных окон среды, а также с системой помощи и окном Evaluate (о нем речь пойдет в главе, посвященной отладке), связана своя подсказка, отражающая специфику его использования. Подсказка может иметь "подводную часть", высвечиваемую при достаточно длительном (около 2 секунд) нажатии клавиш Alt или Ctrl - здесь содержатся краткие названия некоторых команд, вызываемых с использованием нажатой клавиши. Система подсказок особенно полезна новичкам, им советуем в секунды ожидания почаще опускать глаза к нижнему обрезу экрана.

Одно из очень полезных свойств среды Турбо Си состоит в наличии контекстно-зависимой системы помощи. При нажатии пользователем в любой точке среды клавиши F1 на экране появляется окно, в котором приведены короткие, но емкие, объяснения по поводу того, что, по мнению системы помощи, может заинтересовать пользователя в этом месте. Так, нажав F1 в окне редактора, вы получите коротенькое (всего 7 "страничек") описание команд редактора. Находясь в меню, вы сможете прочитать объяснения по поводу той позиции меню, на которой установлена подсветка в момент нажатия F1. В окнах "Message" и "Watch" - свои объяснения, и так далее.

Еще одна великолепная возможность системы помощи - помощь по языку Си, также контекстно-зависимая. Для вызова этой системы нужно в окне редактора подвести курсор к интересующему вас элементу программы

и нажать **Ctrl-F1**. Если курсор установлен на ключевом слове Турбо Си, будет выведена информация о назначении и синтаксисе соответствующей этому слову конструкции языка. Если курсор на стандартном для Турбо Си идентификаторе, будет приведено описание идентификатора, краткие пояснения и имена стандартных **header**-файлов, в которых он объявлен. В остальных случаях сообщается о том, что система ничего конкретного сказать не может и предлагает два основных раздела помощи - по ключевым словам и по **header**-файлам.

Здесь пора обратиться к существенному свойству системы помощи: большинство экранов, предъявляемых системой, имеют ссылки на понятия, родственные объясняемому или как-то с ним связанные, иначе говоря, на то, что еще может заинтересовать читателя данного экрана системы помощи. Ссылки эти разбросаны по тексту в виде выделенных иным цветом слов, и набор этих слов ведет себя как меню: перемещаться от слова к слову можно стрелками и клавишами **Home** и **End**, нажатие **Enter** приводит к выводу на экран объяснений о выделенном понятии. Предыдущий текст с экрана стирается, но не забывается. Нажатие **Alt-F1** приведет вас обратно к нему, еще **Alt-F1** - к тому, который был показан перед ним, и так далее (так запоминается до 20 текстов помощи). Более того, покинув систему помощи нажатием **Esc**, вы имеете возможность вернуться к последнему из просмотренных экранов все тем же способом - **Alt-F1** - и от него идти дальше по системе (вперед или назад).

Наконец, если вы находитесь в системе помощи, в любой момент можно получить нажатием **F1** оглавление этой системы и выбрать из него тот раздел, который вас интересует.

2. РЕДАКТОР

2.1. ХАРАКТЕРИСТИКА РЕДАКТОРА

Если оценивать сравнительную полезность составных частей среды Турбо Си тем временем, которое пользователь проводит в общении с каждой из них, самой нужной подсистемой, вне сомнения, окажется встроенный редактор. И не только потому, что большинство людей набирают программы медленнее, чем эти же программы переваривает компилятор Турбо Си.

Возможность компилировать и выполнять программу, имея перед глазами ее текст, и при необходимости вносить любые изменения, не вызывая для этого никаких других инструментов, оказалась чрезвычайно полезной. Об этом говорит популярность Турбо Си как интегрированной системы в противовес другим Си-системам, представляющим собой разрозненные наборы программ, вызываемых из не слишком дружественной среды ДОС.

Пользователь таких систем, конечно, может возразить, что он работает в своем любимом редакторе и никто не может ему навязать другой, а в Турбо Си программист цепями прикован к редактору, данному ему небесами в лице фирмы **Borland International**. Не слишком убедителен этот довод: ведь помимо того, что встроенный редактор Турбо Си достаточно мощен и удобен для написания программ, работа с редактором и компилятором одновременно - нечто совершенно иное, чем просто редактирование, и сравнивать эти две вещи - все равно, что говорить о преимуществе мясорубки перед кухонным комбайном с дюжиной функций (пусть даже мясорубка в комбайне похуже).

В самом деле, кто кроме пользователя среды "редактор-компилятор", после исправления одной из 20 найденных компилятором ошибок может позволить себе не заниматься остальными девятнадцатью, а вместо этого запустить компилятор еще раз, чтобы посмотреть, не исчезло ли вместе с этой ошибкой две-три наведенных?

Справедливости ради стоит отметить, что опытный приверженец командной строки даже в условиях постоянных прыжков от редактора к компилятору и обратно запросто сможет обогнать ленивого, едва знакомого с редактором Турбо Си, даже при прочих равных условиях. Вывод: изучайте свой инструмент - редактор Турбо Си - так, чтобы, как говорят англичане, все команды были у вас на кончиках пальцев.

Назначение этой главы - помочь в изучении редактора. Если посадить за компьютер любого хоть чуть-чуть знакомого с IBM PC человека и предложить ему поработать в редакторе Турбо Си, ему не понадобится много времени, чтобы обнаружить, что стрелки работают так, как предполагается, клавиши Backspace, Enter, Ins, Del и им подобные тоже выполняют привычные функции, и так далее - в этом все редакторы одинаковы (возможно, есть исключения, но в голову что-то они не приходят).

Чуть более подкованный пользователь усвоит, что Ctrl-PgDn перемещает курсор в конец текста, тогда как Ctrl-End - в конец текущего экрана, Ctrl-Y удаляет строку, а Ctrl-T удаляет слово (точнее говоря, то, что редактору кажется словом). Команды такого рода здесь описывать не будем. Вместо этого уделим внимание другим командам, а именно тем, которые, по наблюдениям автора, в ходу далеко не у всех, что отнюдь не говорит об их бесполезности. Но перед тем, как начать знакомство с командами, посмотрим на экран.

Верхняя строка окна редактора Турбо Си содержит информацию о состоянии редактора, а именно:

- позицию (номер строки и столбца) файла, на которой находится курсор;
- активные режимы редактирования (подробнее о них речь ниже);
- имя редактируемого файла, причем если текст файла не совпадает с текстом, записанным на диске, то перед его именем стоит звездочка.

2.2. РЕЖИМЫ РЕДАКТИРОВАНИЯ

Первый из тех, что обозначаются в строке состояния, называется Insert и говорит о... правильно, о том, что вводимые символы вставляются в текст перед символом, под которым стоит курсор в момент нажатия клавиши. Отключается и включается обратно режим Insert, как обычно, клавишей Ins. Есть, впрочем, и другой способ - нажать Ctrl-V, но это уже информация, забывать которой голову не стоит.

Режим Indent весьма удобен для тех, кто любит вводить программы "лесенкой". Когда в режиме Indent нажимают Enter, появляется новая строка и курсор автоматически устанавливается в позицию, с которой начинается текст предыдущей строки. Переключатель режима Indent - Ctrl-O I.

Следующий режим - Tab. В этом режиме нажатие кнопки "табуляция" приводит к передвижению курсора вправо на 1 - N позиций (N - число, установленное командой "Options\Environment\Tab size", по умолчанию 8; точное число пропускаемых позиций можно посчитать, но это совершенно неинтересно). Все символы текущей строки, которые находятся справа от курсора (включая тот, что над ним), также сдвигаются вправо. Стоит отметить, что, несмотря на видимость заполнения пропущенных позиций пробелами, на диске в этом месте будет стоять символ табуляции, который интерпретируют далеко не все редакторы.

Кроме того, при движении курсора вверх невидимых в редакторе Турбо Си символов табуляции могут возникать неожиданные скачки курсора. Тем, кто во что бы то ни стало хочет пользоваться клавишей табуляции в ее истинном предназначении, стоит сначала освоиться в режиме Tab и в дальнейшем ничему не удивляться.

Остальным расскажем о том, как работает эта клавиша табуляции, когда режим Tab отключен (переключатель режима Tab - Ctrl-O T). Курсор тоже будет передвигаться вправо, но теперь он ориентируется не на размер табуляции, а на предыдущую строку. Остановки курсора - под первым символом каждого из составляющих ее слов (словом здесь считается любая последовательность символов, обрамленная пробелами).

Звучит все это, боюсь, не слишком понятно, но стоит только попробовать - и все сразу станет ясно. Любителям "лесенки" эта возможность тоже должна понравиться. Добавим, что начальные символы строки, пропускаемые в этом режиме, заменяются не символами табуляции, а настоящими пробелами, так что курсор ведет себя вполне пристойно.

Режим Fill. Режим этот явно представляет собой тяжелое наследие старых времен (работа только с дискетами, а винчестер на 10 мегабайт - почти недоступная роскошь), когда сэкономить два десятка байт от файла в 2-3 К было ошутимой победой. В режиме Fill (он работает только одновременно с режимами Indent и Tab) эти байты экономятся так: при заведении новой строки курсор, как положено в режиме Indent, устанавливается под первой буквой предыдущей строки; головная (пропущенная) часть строки при этом заполняется смесью символов табуляции и пробелов так, чтобы она занимала как можно меньше байт на диске (отсюда второе название режима - *Optimal Fill*, что означает "*оптимальное заполнение*").

Включается и выключается режим Fill нажатием Ctrl-O F. Еще раз напомним, что режим Fill включается, только если включены режимы Insert и Tab, соответственно выключается автоматически при отключении любого из них.

Режим Unindent. Когда включен этот режим, клавиша Backspace начинает выполнять дополнительную функцию: если слева от курсора текущая строка не содержит символов, отличных от пробела, при нажатии Backspace курсор (а за ним и правая часть строки) смещается в позицию, с которой начинается текст предыдущей строки. Более того, если после этого еще раз нажать Backspace, повторится то же действие, но уже относительно ближайшей сверху строки, выступающей влево по сравнению с предыдущей, еще раз Backspace - относительно следующей и так далее.

Таким образом, режим Unindent опять-таки предназначен для любителей записывать программы "лесенкой". Из всего этого можно сделать вывод, что создатели редактора сами принадлежат к этой славной части цивилизации программистов и предлагают всем к ней присоединиться. Поскольку они делают это весьма ненавязчиво (принципиальные "антилессеночники" всегда могут отключить ненавистные режимы), их можно только поздравить и поблагодарить.

Прежде, чем перейти к рассмотрению избранных команд редактора, скажем пару слов об имеющихся ограничениях.

Во-первых, строка не может быть длиннее 248 символов (кстати, ширина окна редактора - 77 символов). Всего строк в редактируемом файле может быть сколько угодно. Ограничения диктуются, во-первых, архитектурой IBM PC (весь файл может занимать не более 1 сегмента памяти - 64 Кбайта с учетом служебной информации редактора), во-вторых, имеющимся объемом оперативной памяти: если достаточно большая часть памяти занята другими программами, то редактор среды может отказаться работать и значительно раньше.

При появлении в строке состояния сообщения редактора о недостатке памяти (кстати, по мере приближения к пределу редактор периодически об этом предупреждает) следует сохранить файл и подумать о причинах отказа. В первом случае файл придется разбить на более мелкие части, во втором - перезагрузиться или освободить память каким-либо другим способом (возможно, выбросить из оперативной памяти не самые нужные резидентные программы).

Теперь о символах, отображаемых редактором. Если вы попытаете загрузить в редактор какой-нибудь нетекстовый файл (например, EXE-файл), вы увидите, что часть символов выделена другим, непривычным цветом. Это управляющие символы с кодами от 1 до 31. Обычно набирать их вручную не нужно, но если вдруг такая нужда появится (например, очень хочется поставить символ перехода к новой странице печати), можно нажать **Ctrl-P** и затем - символ, соответствующий требуемому. Соответствие определяется обычным для ДОС способом: **Ctrl-A** имеет код 1, **Ctrl-B** - код 2, и так далее.

Наконец, можно поговорить и о командах редактора.

Во-первых, о перемещении по тексту. Всем известно, что для этого используются клавиши дополнительной клавиатуры - стрелки, **PgUp**, **PgDn**, **Home** и **End**. Не все знают, однако, что в редакторе среды Турбо Си имеется еще одна возможность проделывать те же действия, связанная с использованием клавиши **Ctrl** и клавиш из левой части основной клавиатуры.

Так, **Ctrl-S** выполняет ту же функцию, что "стрелка влево", **Ctrl-D** - "стрелка вправо", **Ctrl-X** и **Ctrl-E** - соответственно вниз и вверх (запомнить довольно просто, взглянув на расположение этих клавиш). **Ctrl-A** и **Ctrl-F** выполняют перемещение по словам (аналог **Ctrl-стрелок влево и вправо**), **Ctrl-C** и **Ctrl-R** - листание экрана вниз и вверх.

Наконец, команды **Ctrl-Z** и **Ctrl-W** никого не дублируют, а сдвигают экран вниз и вверх на одну строку, не меняя позиции курсора.

Возникает вопрос: зачем все эти клавиши, если для большинства из них есть гораздо более удобные аналоги? Вместо ответа - задача: даны 10 соседних строк, требуется удалить из каждой все символы, начиная с 51-го. Не спешите демонстрировать владение командой **Ctrl-Q Y** (*удалить все символы справа от курсора*) и цифровой клавиатурой: оцените нажатие **Ctrl-Q Y X** (клавишу **Ctrl** отпускать не нужно!) десять раз подряд. Может быть, ты, о читатель, относишься к той части человечества, которой этот способ не покажется гораздо более удобным и быстрым? Аналогичные примеры можно придумать и для прочих дополнительных "навигационных" клавиш.

Раз уж речь пошла о перемещении по тексту, нельзя не упомянуть о возможности отмечать различные места текста для последующего мгновенного перевода курсора в них. Последовательно нажав **Ctrl-K** и *n*, где *n* - номер отметки (всего можно ставить до четырех отметок, так что *n* изменяется от 0 до 3), вы помечаете позицию (номер строки и столбца в файле), в которой находится курсор. Нажав затем **Ctrl-Q n** в любом месте текста, вы вернете курсор в отмеченную позицию. Очень удобная и явно недооцениваемая пользователями возможность.

Самая, пожалуй, загадочная команда редактора Турбо Си - это **Ctrl-Q L**, что системой помощи объясняется как "**Restore line**" (*"Восстановить строку"*). Первое, что приходит в голову при таком объяснении, - можно вернуть строку, удаленную командой **Ctrl-Y**. Попробуем! Не получится... Похоже, команда не работает... Работает, конечно, только не совсем так, как мы от нее ожидали.

Вот для чего используется команда **Ctrl-Q L**. Если вы начали редактировать строку и вдруг осознали, что делаете что-то не то, у вас есть шанс вернуть старое содержимое строки. Нажмите **Ctrl-Q L**, и строка появится на экране в том виде, какой она имела в момент последней установки на

нее курсора. Внимание: строку можно восстановить только до того, как курсор ее покинет!

Перейдем теперь к командам работы с блоками текста. Трудно представить себе, что можно работать в редакторе Турбо Си и не знать что такое блок и как его скопировать, переместить или уничтожить. Менее популярны команды записи блока на диск, чтения блока с диска и печати блока. Совсем мало используются команды сдвига блока влево или вправо (забавно, что об этих двух командах даже не упомянуто в тексте помощи по редактору - надо полагать, разработчики help-системы о них тоже не знают?). Итак, поподробнее.

Запись блока на диск - Ctrl-K W. Предлагается задать имя файла, в который будет записан текст, выделенный на экране. Если файл с заданным именем уже существует, старое его содержимое погибает (как обычно, пользователя перед этим спросят, уверен ли он в своем решении). Обратная операция - Ctrl-K R. Введя имя файла, вы получите его содержимое вставленным перед курсором.

Если вы не уверены в точном написании имени требуемого файла или не желаете его набирать полностью, можно использовать маску и выбрать файл из списка-меню. На всякий случай скажем, что при помощи команды Ctrl-K R можно считать произвольный файл, а не только тот, что был создан командой Ctrl-K W.

Печать блока - Ctrl-K P. Принтер к моменту вызова команды рекомендуется включить, бумагу вставить и перевести аппарат в режим On-line. Если это не сделано, а также в случае окончания листа среда выведет сообщение "Error accessing PRN device. Retry or Abort?" Исправьте ситуацию и для продолжения нажмите R. Не хотите продолжать общение с принтером - Esc или A.

Сдвиг блока вправо - Ctrl-K I (I - от слова *Indent*, что означает "*сделать отступ*"), влево - Ctrl-K U (*Unindent*, "*сделать обратный отступ*"). Команды, радующие сердце любителя "лесенки", особенно в случае, когда на другой уровень вложенности нужно перевести достаточно большую часть текста.

Наконец, если на экране маячит выделенный цветом блок и это вас, понятное дело, нервирует, есть гораздо более удобный способ подсветку удалить, чем создавать "пустой" блок (Ctrl-K K и ниже по тексту - Ctrl-K B; так поступают многие темные люди). Не дергая курсором, нажмите Ctrl-K H. Подсветка исчезнет. Еще раз Ctrl-K H - подсветка вернулась, и блоком снова можно манипулировать.

Без поиска строк редактор - не редактор, а без поиска с заменой редактор - плохой редактор. Среда Турбо Си обладает неплохим редактором. Более того, поиск и замена реализованы в нем, может быть, не для всех привычно, но довольно гибко и удобно.

Напомним, команда поиска - Ctrl-Q F, поиск с заменой - Ctrl-Q A. Приглашение на ввод образца поиска содержит в качестве предположения образец из прошлого обращения к поиску. Как и в других подобных случаях, можно сразу набирать новый образец, а можно нажатием клавиши на вспомогательной клавиатуре оставить данный вариант и его отредактировать. То же относится к строке замены. Это все вполне очевидно, а вот приложение к вводу образцов - ввод опций поиска/замены - довольно интересно, к тому же не общезвестно. Итак, опции поиска и замены.

Сначала - что будет, если не задано никаких опций. Поиск ведется от позиции курсора до конца файла. Будет найдена и предъявлена первая строка, соответствующая образцу (с различением строчных и прописных букв). Для повторения поиска с тем же образцом нажимают Ctrl-L. Если производится поиск с заменой, ищутся сразу все подходящие строки, и о каждой из них пользователю будет задан вопрос "Replace? (Y/N):" ("Заменить?"). Утвердительный ответ - Y, отрицательный - N. Для того, чтобы остановить процесс поиска с заменой, в ответ на вопрос нажмите Esc (эстеты могут нажимать Ctrl- U, здесь это безразлично). Продолжить поиск и замену с теми же образцами - тоже Ctrl-L.

А теперь - сами опции, задание каждой из которых немного изменяет вышеприведенную картину. Под словом "поиск" здесь понимается также и поиск с заменой.

B - поиск "назад". Поиск начинается с позиции курсора, но происходит в направлении начала файла.

G - глобальный поиск. Поиск начинается с курсора и продолжается до конца файла (как видно, отсутствие опций эквивалентно заданию опции G).

L - поиск внутри блока. Если в этот момент есть выделенный блок, поиск(-замена) начнется с начала блока и закончится в его конце. Если блока нет, то и поиска не будет.

p (p - десятичное число). Поиск каждого p-го вхождения образца. Все остальные вхождения пропускаются.

U - поиск без различия строчных и прописных букв.

W - поиск слов.

Соответствующими образцу считаются только последовательности символов, представляющие собой в точности образец, обрамленный разделителями (разделителем здесь считается все, кроме латинских букв и цифр, в том числе и буквы кириллицы).

N - замена без вопросов (на поиск без замены опция не влияет). Пользователь имеет возможность понаблюдать, как без его участия редактор заменяет так, как ему указано, все, что удастся найти. Будьте внимательны, чтобы успеть в случае чего прервать этот процесс нажатием Esc или Ctrl-U.

Опции могут задаваться в любой комбинации, подряд или через пробел, или вообще через какие угодно символы - все, что не является знаком опции, здесь игнорируется.

На десерт опишем пару очень любопытных команд - Ctrl-Q [и Ctrl-Q]. Работают они так: если подвести курсор к открывающей скобке и нажать Ctrl-Q [, курсор переместится к закрывающей скобке, соответствующей исходной. Аналогично Ctrl-Q] переходит от закрывающей скобки к соответствующей ей открывающей.

Такого рода сервис, обычный для редакторов, использующихся при программировании на ЛИСПе с его нагромождением скобок, довольно часто оказывается полезным и в Си - возможно, это следствие характерного для любителей Си стремления затолкать в одно выражение как можно больше действий. Команды Ctrl-Q [и Ctrl-Q] работают для всех трех типов скобок, имеющих в Си: круглых (), квадратных [] и фигурных (операторных) { }.

Кроме скобок, они работают также с комментариями - символы /* считаются открывающей скобкой. */ - закрывающей. Но и это еще не все: подведя курсор к одинарной (') или двойной (") кавычке и, нажав

Ctrl-Q], вы увидите, что курсор переместился к ближайшей из (таких же) кавычек, предшествующих данной. Соответственно, Ctrl-Q [переходит к следующей по тексту кавычке.

3. КОМПИЛЯТОР

3.1. КАК ПОЛУЧИТЬ ВЫПОЛНЯЕМУЮ ПРОГРАММУ

Давайте начнем главу о работе с компилятором среды Турбо Си с обсуждения того, каким образом из текста на Си получается выполняемая программа, из каких шагов состоит этот процесс и какие файлы при этом используются и порождаются. Возможно, многим эта тема покажется неинтересной, так как все ясно и без разговоров. Тем не менее без такого рода знаний очень трудно ориентироваться в происходящем на экране и на диске, а уж при возникновении каких-либо проблем обычно приходится теревить знатоков (пожалеем этих несчастных, вынужденных по двадцать раз на день отвечать на дурацкие вопросы типа "почему у меня не идет?" - понятно, по какой причине многие из них норовят работать по ночам). Приведа эту необходимую информацию, можно будет перейти к основному содержанию главы - работе с опциями компилятора и борьбе с ошибками и предупреждениями. Предварительно - одно терминологическое замечание. В нижеследующем тексте слова "компилятор" и "транслятор" употребляются как синонимы. То же относится и к словам "компиляция" и "трансляция".

С точки зрения пользователя Турбо Си весь путь от момента, когда текст программы готов к компиляции, и до запуска ее на выполнение разбит на две части. Первая - компиляция, вторая - сборка выполняемой программы из отдельных откомпилированных файлов и библиотек объектных модулей. Ход процесса зависит от того, сколько модулей (отдельно транслируемых файлов) занимает программа. Сначала рассмотрим простой случай, когда вся программа содержится в одном файле.

На входе компилятор имеет текст программы, находящийся в окне редактора. Встречая в тексте директивы `#include`, компилятор (точнее говоря, препроцессор, работающий одновременно с ним) ищет указанные в них файлы в текущем директории, затем - в директориях, заданных в позиции "Include directories" меню "Options/Directories". Отметим, что привычное для Си соглашение о том, что файлы, имена которых в директиве `#include` заключены в угловые скобки, ищутся в некотором "стандартном" месте (директории), а если имя заключено в двойные кавычки, то в текущем директории, а затем в стандартном, в Турбо Си не выполнено: стандартные файлы сначала все-таки ищутся в текущем директории, а "обычные" - только в текущем. Текст программы со вставленными в него в местах соответствующих директив `#include` файлами транслируется как единое целое.

В случае отсутствия ошибок при трансляции компилятор создает объектный файл, имеющий имя исходного и расширение `.OBJ`. В файле записан сгенерированный для программы код и обозначены ссылки на внешние по отношению к программе объекты (для одномодульной программы все эти объекты должны быть стандартными функциями или переменными Турбо Си).

Следующий этап - сборка программы, называемая на английском языке *link*, а на русском жаргоне - "линковка" (сделаем уступку жаргону и будем программу сборки в дальнейшем называть линкером). На входе линкера - объектный файл и библиотеки

Турбо Си, содержащие коды всех стандартных функций и информацию о памяти, отводимой под стандартные переменные. Линкер проверяет наличие в библиотеках внешних функций и переменных, на которые ссылается программа, и составляет из объектного файла программы и используемых библиотечных объектов выполняемую программу, которую и записывает на диск в файл с именем опять-таки исходного текста и расширением EXE. В этом файле собраны воедино коды всех функций, участвующих в выполнении программы, определено месторасположение глобальных переменных, а бывшие ссылки на "внешние" имена заменены ссылками на объекты, носящие эти имена. Процесс замены таких ссылок называется редактированием внешних связей, отсюда еще одно русское название линкера, красивое и торжественное, хотя немного длинное - "редактор внешних связей".

Файл типа EXE уже может быть выполнен как из среды Турбо Си командой "Run", так и из ДОС. При запуске программы из среды Турбо Си возможна отладка при помощи встроенного отладчика (возможность эта, впрочем, зависит от режима, в котором транслировалась программа, об этом ниже в этой главе, а также в главе "Отладка в интегрированной среде"). Компиляция и (в случае успеха) сборка программы могут быть проведены одной командой, называемой "Make", или ее разновидностью "Build".

Порядок действий при подготовке к запуску программы, состоящей из нескольких модулей, такой же. Отличие в том, что перед запуском линкера необходимо оттранслировать все модули, составляющие программу, и поместить полученные объектные файлы в текущем директории, либо в директории, указанном в позиции "Output directory" меню "Options/Directories". Это может быть сделано как вручную, так и автоматически командами "Make" или "Build". Нужно лишь составить файл, управляющий работой этих команд, так называемый файл проекта (project file). Подробности составления файла проекта, а также подробное описание действий по его обработке приводятся в главе 5 "Управление проектом". Здесь важно лишь, что в файле проекта перечислены все составляющие программу модули и (необязательно) библиотеки пользовательских объектных программ.

После того, как все объектные файлы получены, программа может быть собрана. Линкер также использует файл проекта: во-первых, для того, чтобы знать, какие OBJ-файлы нужно искать, во-вторых, для поиска внешних по отношению к программе объектов не только в стандартных библиотеках Турбо Си, но также и в библиотеках, указываемых здесь пользователем. В отличие от одномодульного варианта, на этом шаге могут возникнуть различные трудности, описанные в следующей главе - "Сборка выполняемой программы". Если эти трудности преодолены, результат - тоже файл с расширением EXE, имя же его совпадает с именем файла проекта.

Все OBJ- и EXE-файлы записываются на диск в директорий, указанный в позиции "Output directory" меню "Options/Directories", или, если там ничего не указано, в текущий директорий.

3.2. КОМАНДЫ, ЗАПУСКАЮЩИЕ РАБОТУ НАД ПРОЕКТОМ

Команды, запускающие компилятор и линкер для различных действий, собраны в меню "Compile", которое сейчас коротко рассмотрим.

"Compile to OBJ" (Alt-F9). *Компилировать в OBJ-файл.* В этой позиции записано также имя объектного файла, который будет создан при выборе этой команды. Объектный файл имеет расширение .OBJ и имя, совпадающее с именем файла, заданного в позиции "Primary C file:" этого же меню (см. ниже). Если же в ней не задано никакого файла, имя для OBJ-файла выбирается по названию файла, находящегося в данный момент в окне редактора.

"Make EXE file" (F9). *Сделать EXE-файл.* Здесь также записано имя файла. Вызов этой команды (называемой еще авто-Make, в отличие от утилиты make, которая работает несколько по-другому - см. главу 7 "Работа из командной строки") приводит к созданию - по крайней мере, к попытке создания - файла с выполнимой программой. Для этого получают OBJ-файлы всех составляющих программу модулей и вызывается редактор связей, результатом работы которого (если нет ошибок, конечно) и является выполнимый EXE-файл. Название этого файла записано в рассматриваемой позиции меню.

Имя для EXE-файла команда авто-Make выбирает следующим образом:

- если в позиции меню "Project/Project name" задан файл проекта, берется его имя; в противном случае:
- если в позиции меню "Compile/Primary C file" задано название файла, берется имя из этого названия (исключаются путь и расширение); в противном случае:
- берется имя файла, находящегося в окне редактора.

Отметим, что для обработки многомодульных программ команде Make обязательно требуется файл проекта, для программы из одного модуля он необязателен.

Очень важным свойством команды Make является тот факт, что в случае отсутствия каких-либо OBJ-файлов, нужных для сборки выполнимой программы, она самостоятельно вызывает компилятор для трансляции соответствующих файлов. Более того, если какой-нибудь OBJ-файл имеется, но не соответствует исходному тексту, из которого получен (точнее говоря, если дата последнего изменения исходного текста больше, чем дата создания OBJ-файла), такой OBJ-файл также будет создан заново.

Более подробно о работе команды Make, меню "Project", файлах проекта и прочих вопросах, связанных с построением многомодульных программ, рассказано в главе "Управление проектом".

"Link EXE file". *Собрать EXE-файл.* Эта команда - вызов линкера. Предполагается, что все файлы, которые должны быть на входе редактора связей, сформированы и расположены в соответствующих директориях (см. ниже обсуждение меню "Options/Directories", а также главу "Управление

проектом"). Эта команда хотя и не столь часто используется, как ее родственные *Compile* и *Make*, тем не менее иногда очень удобна, например, для включения в ЕХЕ-файл или исключения из него отладочной информации, в случае изменения библиотек и в некоторых других.

"*Build all*". *Все перестроить*. Эта команда родственна команде *Make*, она тоже на входе имеет файл проекта (в случае многомодульной программы) либо текст единственного файла программы, тоже вызывает (возможно, не один раз) компилятор, а затем линкер и точно тем же образом выбирает имя создаваемого ЕХЕ -файла. Разница единственная: все файлы, составляющие программу, перекомпилируются без всяких условий. Команда "*Build*" полезна, например, в случае, когда изменены опции компилятора и требуется распространить новые установки на все файлы программы.

"*Primary C file*". *Имя основного файла*. Как говорилось выше, в случае, если не задан файл проекта, команды *Compile*, *Make* и *Build* компилируют текст на Си, записанный в файле, указанном в этой позиции. Наличие возможности транслировать файл, не обязательно находящийся в окне редактора, иногда бывает очень удобно. Представьте такую картину: вы запускаете компиляцию и получаете сообщение об ошибке в одном из файлов, включенных в текст директивой *#include*. Естественно загрузить этот файл в редактор и исправить ошибку. После этого придется вернуться в исходный файл и запустить компилятор по новой. Если же файл объявлен основным, возвращаться будет не нужно, что сэкономит несколько секунд и еще немного душевных сил.

"*Get info*". *Получить информацию*. Эта команда выводит на экран окно с информацией о редактируемом файле, о последних компиляции и запуске, а также о состоянии среды. Подробнее:

"*Current directory*:". *Текущий директори*. Нужно ли объяснять?

"*Current file*:". *Текущий файл*. Название файла (полное, с дисководом и путем), находящегося сейчас в окне редактора.

"*File size*:". *Размер файла*. Длина файла в байтах. Здесь же указана максимальная длина файла, с которым может работать редактор (от 20000 до 65534, в зависимости от заданной при помощи утилиты *TCINST* длины буфера редактора). Когда реальная длина файла приближается к максимальной, редактор начинает выдавать сообщения вида "*WARNING: nnn byte(s) left*" - "*Предупреждение: осталось столько-то байт*". После выдачи этого сообщения можно нажать *Esc* и продолжать редактирование, но недолго - до исчерпания буфера редактора, о чем известит сообщение "*ERROR: Out of space. Press Esc*" ("*ОШИБКА: Исчерпано пространство. Нажмите Esc*"). При появлении этого сообщения приходится разбивать файл на части или подыскивать другой редактор, переходя на работу с Турбо Си из командной строки.

"*EMS usage*:". *Использование расширенной памяти*. Если ваш компьютер оборудован платой расширения оперативной памяти и загружен драйвер расширенной памяти, поддерживающий стандарт *EMS* (*Extended Memory Specification*), то у вас есть возможность освободить часть оперативной памяти, выслав буфер редактора в эту самую расширенную память. Собственно говоря, среда Турбо Си, обнаружив драйвер, все остальное делает сама, сообщив вам лишь, сколько килобайт расширенной памяти занято под буфер редактора.

"*Lines compiled*:". *Всего скомпилировано строк*. Общее количество строк, подвергшихся трансляции при последнем запуске компилятора. Учи-

тываются как строки основного файла, так и строки файлов, включенных в него директивой `#include`.

"Total warnings:". *Всего предупреждений*. Тоже относится к последней трансляции.

"Total errors:". *Всего ошибок*. И это тоже о последней трансляции.

В правой нижней части окна находятся слова, описывающие состояние среды: статус выполняемой программы (варианты: "Программа не загружена", "Программа выполняется", "Программа завершилась"), код завершения последнего запуска (это тот самый код, который можно установить в программе при помощи функции `exit`; он же - значение, которое вернула функция `main`, если вы описали ее как `int`) и объем свободной оперативной памяти.

Перейдем теперь к рассмотрению опций компилятора, которые скрыты в меню **"Options/Compiler"**.

"Model". *Выбор модели памяти*. В этом месте можно было бы завести разговор о том, что такое модель памяти, какие модели памяти существуют и из каких соображений эту самую модель выбирать.

Однако нас останавливают два соображения. Первое: "Руководство пользователя Турбо Си" освещает все эти вопросы с исчерпывающей полнотой, повторяться или делать это хуже не хочется, а сделать лучше - задача, как говорится, архисложная. Второе: и авторы, и большинство их знакомых всю свою жизнь в Турбо Си используют одну-единственную модель памяти (`small` или `large` - в зависимости от вкусов), и все чувствуют себя прекрасно. Это не значит, конечно, что о моделях памяти знать ничего не нужно, но... здесь мы возвращаемся к соображению 1.

Рекомендем читателям, еще не выбравшим себе любимую модель памяти или не интересующимся такими мелочами, установить модель `Large` - отсутствие каких-либо специфических забот при разработке программ среднего размера гарантировано. Желаяющим же поэкспериментировать с разными моделями следует обратиться к "Руководству пользователя" или по крайней мере помнить, что все модули многомодульной программы должны быть оттранслированы в одной модели, а в случае использования пользовательских библиотек объектных программ важно соответствие между библиотекой и используемой моделью памяти.

"Defines". *Определение препроцессорных переменных*. Помимо обычного для Си определения препроцессорных переменных, имеющего вид предложения в тексте программы

```
#define имя  
или  
#define имя = значение ,
```

Турбо Си предлагает еще один способ - из меню интегрированной среды. В позиции **"Defines"** могут быть записаны одно или несколько таких определений. Каждое из них выглядит обычно, за исключением того, что опускается слово `#define`. Разделяются определения точкой с запятой. Если есть необходимость внутри определения использовать точку с запятой, перед ней, как это привычно для Си, записывается символ `'\'` (обратная косая черта). Задание здесь непустого набора определений приводит к тому, что перед трансляцией каждого файла в начало его будет "добавляться" (конечно, не физически, а, так сказать, в идеальном смысле) соответствующее

число директив `#define` с заданными определениями. На всякий случай приведем пример. Предположим, вы задали в позиции "Defines" следующую строку:

```
debug; semicolon = '\;'; hardware = 'AT'
```

Для каждого из файлов, который вы будете транслировать в дальнейшем (конечно, только до момента, когда вы данную строку измените), это будет эквивалентно присписыванию в голову файла следующего текста:

```
#define debug  
#define semicolon = '\;'  
#define hardware = 'AT'
```

"Code generation". Меню опций генерации кода. В этом меню собраны опции, сообщающие компилятору кое-какие детали генерации объектного кода программ. Изучим это меню.

"Calling convention". Тип соглашений о связях. Под соглашениями о связях здесь понимаются правила оформления (в машинных командах, конечно) входа в функцию и выхода из нее. Переключатель имеет два положения, соответствующие двум стандартам соглашений: общепринятому для Си и обычному для Паскаля. Пусть наличие возможности генерировать коды функций в паскалевской манере не наводит вас на мысль о возможности вызова подпрограмм Турбо Паскаля из функций Турбо Си - такой возможности нет, как нет OBJ-файлов в Турбо Паскале. Паскалевские соглашения могут понадобиться при программировании на Турбо Си, когда нужно обратиться к подпрограмме, написанной на каком-либо другом языке (например, ассемблере или, скажем, Микрософт Паскале), при этом нужно быть уверенным, что вызываемая подпрограмма соблюдает паскалевские соглашения о связях.

Вообще связь Турбо Си с другими языками, равно как и использование паскалевских соглашений в рамках Турбо Си, - тема для отдельного разговора. Проблемы, здесь возникающие, довольно специфичны, требования к квалификации программиста высоки, и в книге об использовании среды Турбо Си разбор такого рода вопросов был бы неуместен. Тем же, кому не терпится совокупить Турбо Си с чем-либо другим, советуем обратиться к фирменному руководству по Турбо Си, где вся (или почти вся - зависит от читателя) необходимая информация изложена довольно подробно. Остальным рекомендуем установить переключатель в положение "C" (скорее всего, он там уже находится) и забыть о его существовании.

"Instruction set". Система команд. Имеется в виду тип ЭВМ, для выполнения на которой готовится объектная программа. Не беспокойтесь, Турбо Си для больших машин серии ЕС еще не генерирует. Возможны всего 2 варианта: первый - машина с процессором Intel 8086 или Intel 8088 (иначе говоря, PC или XT), второй - с процессором Intel 80186 или Intel 80286 (то есть AT). Вторая система команд является расширением первой, так что программы, оттранслированные в режиме "8088/ 8086", смогут выполняться на машинах типа AT, но не наоборот. Так что если вы или ваши пользователи работают на PC (есть ли такие?) или XT (о, таких - тьма), держите переключатель в положении "8088/ 8086". В противном случае естественно генерировать в коды AT (положение "80186/80286"). Однако если объявится пользователь со старомодной техникой, придется посоветовать ему сменить аппаратуру (вариант: за отдельную плату перскомпилировать поставляемую программу в режиме "8088/8086").

"Floating point". Тип плавающей арифметики. Возможны три варианта. Первый из них таков: ваш компьютер имеет арифметический сопроцессор (наличие сопроцессора легко установить, например, программой SI из набора Питера Нортон, либо аналогичной командой системы PCSTOOLS), вы желаете выполнять операции над вещественными данными и о других машинах не задумываетесь. В этом случае годится установка переключателя в положение "8087/80287". На машине без сопроцессора скомпилированные в этом режиме программы работать будут, но лишь "как бы": ввиду того, что выполнять команды вещественной арифметики некому, их никто выполнять и не будет (к сожалению, никаких сообщений об отсутствии сопроцессора, как сделано, например, в Турбо Паскале, программы на Турбо Си не выдают). Что ваша программа при этом насчитает - сказать трудно. Остается лишь посоветовать быть осторожнее.

Более гибко по сравнению с предыдущим второй вариант установки переключателя **"Floating point"** - **"Emulation"**. Программа, скомпилированная в таком режиме, сама определит, доступен ли сопроцессор, и если да, то для выполнения команд вещественной арифметики будет обращаться к нему. Если сопроцессор отсутствует, все действия над вещественными числами будут производиться подпрограммами стандартной библиотеки Турбо Си, естественно, не так быстро, зато правильно. За такую гибкость, конечно, приходится платить - EXE-файл станет примерно на 10 Кбайт больше, чем для той же программы в "чистом" режиме "8087/80287" (увеличение происходит за счет подключения подпрограмм эмуляции сопроцессора).

Наконец, третий вариант - для принципиальных противников вещественных типов данных (их не так уж мало, особенно среди программистов, отскакивающих на прозвище "системный"). Установите переключатель **"Floating point"** в положение **"None"** и не думайте ни о чем. Если же вы поступите принципами и допустите в свою программу вещественные числа, ничего страшного не случится - линкер пожурит вас сообщениями о неопределенных внешних именах устрашающего вида (пример имени **"FIWRQQ"**, остальные в том же духе). Режим этот, конечно, имеет смысл, ибо, как мы уже видели, отсутствие забот о вещественных числах стоит 10 Кбайт памяти, что иногда многовато.

"Default char type". Работа с типом char по умолчанию. Как известно, при упоминании типа char или int можно уточнять область допустимых значений типа модификатором **signed** или **unsigned**, а можно и не уточнять. Для типа int умолчание эквивалентно наличию модификатора **signed**. Вообще-то стандартно так же обрабатывается и тип char, но в Турбо Си имеется способ установить другое соглашение - тип char без модификатора считается беззнаковым (**"Unsigned"**). Появление этого переключателя именно в меню генерации кода можно объяснить тем, что наличие или отсутствие знака заставляет генерировать разные команды преобразования из типа char в тип int.

Впрочем, нам больше нравится другое объяснение - этот переключатель просто не укладывается ни в какое меню, поэтому его ткнули в **"Options\Compiler\Code generation"**... Хорошим тоном будет его не трогать (стандартно переключатель установлен в положение **"Signed"**), а явно писать **unsigned** в каждом случае, где это требуется, там же, где существенно наличие знака, - **signed**. Время, которое может уйти на нахождение ошибок, связанных с чсхардой в использовании переключателя **"Default char type"**, по-видимому, стоит дороже, чем удовольствие от такого поиска.

"Alignment". Выравнивание. Оперативная память IBM PC разделена на слова, каждое из которых состоит из 2 байтов. Тем не менее данные, занимающие те же 2 байта, но расположенные поверх границы машинного слова (то есть имеющие нечетный адрес) могут быть считаны и записаны теми же машинными командами, что и "нормальные" слова. Разница в том, что для слов с нечетными адресами команды чтения и записи работают чуть медленнее.

Пользователь Турбо Си имеет выбор: располагать все свои словные данные (то есть данные всех базовых типов, кроме символьного; "словными" считаются и данные, размер которых больше одного слова) с начала машинного слова (при этом возрастет скорость работы программы, но память может быть распределена менее эффективно за счет "дырок" длиной в 1 байт, приходящихся на данные, занимающие нечетное число байт), либо располагать данные плотно, но заплатить за это уменьшением скорости. Первая возможность реализуется установкой переключателя "Alignment" в положение "Word", вторая - в положение "Byte". Заметим, что обычно разница в скорости, да и в расходе памяти между этими двумя режимами не так уж заметна, и вспоминают о переключателе "Alignment" обычно в момент, когда исчерпаны возможности по ускорению (или, наоборот, уменьшению объема используемых данных) программы.

"Generate underbars". Генерация символов подчеркивания. Как уже говорилось, компилятор помещает в объектный файл ссылки на все внешние имена, используемые в модуле. Добавим к этому, что в объектном файле также имеется список имен глобальных (доступных извне) объектов, расположенных в этом модуле. Обычно (если переключатель "Generate underbars" находится в положении "On") компилятор приписывает всем этим именам в качестве первой буквы символ '_' (подчеркивание). Естественно, линкер ожидает, что имена глобальных объектов в модулях и библиотеках совпадают с именами, по которым на них ссылаются. В рамках Турбо Си так оно и происходит (все имена, определенные в стандартных библиотеках Турбо Си, тоже начинаются с подчеркивания). Если же вы задумали использовать объектные файлы или библиотеки, изготовленные каким-то другим компилятором, у вас начнутся трудности при сборке программы. Помнить о подчеркиваниях нужно и при программировании ассемблерных подпрограмм для вызова из Турбо Си, и даже при написании ассемблерных вставок в код на Си. Пожалуй, на этих сведениях можно остановиться и отослать всех желающих (по-видимому, их будет не так уж много) к руководству по Турбо Си, а остальным посоветовать не трогать переключатель "Generate underbars", изначально установленный в "On".

"Merge duplicate strings". Отождествление совпадающих строк. Обычное дело - использование в разных местах программы строк (речь идет о константах, задающих "массивы символов"), совпадающих по содержанию. Стандарт Си диктует трактовать такие строки как отдельные объекты, невзирая на идентичность их содержимого. Это значит, что строка будет размножена в памяти ровно столько раз, сколько она появляется в тексте программы. Турбо Си более гибок. Поклонники стандарта могут спокойно работать с переключателем в положении "Off". Те же, кто желает сэкономить немного памяти, имеют возможность хранить каждую строку только в одном экземпляре (положение "On"). Однако это не всегда безопасно. В качестве примера можете запустить маленькую, но поучительную програм-

```
#include <stdio.h>
void main (void)
{ char *str = "string";
  str[3]='a';
  /* ... - предположим, здесь достаточно операторов,*/
  /* чтобы забыть об изменении str */
  printf("string");
}
```

"Standard stack frame". *Стандартная стековая рамка.* Слова, что-то говорящие лишь посвященным. Не будем здесь приводить подробности, интереса для большинства пользователей не представляющие. Все, что достаточно знать об этом переключателе, - положение его "On" позволяет при работе с встроенным отладчиком Турбо Си просматривать стек вызовов функций (команда "Debug/Call stack" или нажатие Ctrl-F3). В положении "Off" такой сервис отсутствует, но программа становится чуть короче.

"Test stack overflow". *Проверка переполнения стека.* При достаточно глубоком рекурсивном вызове функций (это наиболее частая причина; бывают и другие) вполне возможна ситуация, когда стек вызовов процедур (в нем запоминаются адреса возвратов, а также размещаются локальные данные вызванных функций) перестает помещаться в выделенном для него месте (размер этого места определяется моделью памяти). В зависимости от значения переключателя "Test stack overflow" эффект может быть двух видов. Если переключатель в положении "On", программа сообщит о переполнении стека ("Stack overflow") и остановится. Если в положении "Off" - программа начнет вести себя совершенно непредсказуемо, и можно лишь пожелать пользователям, решившего найти ошибку методом "грубой силы" - у автора еще свежи воспоминания о поиске такой "ошибки" в течение двух дней (в качестве слабого оправдания скажу, что программа была большой и сложной, уже находилась в эксплуатации и, естественно, оттранслирована была "по-боевому" - без всяких проверок). Естественно поэтому до окончания отладки держать переключатель в положении "On" - это повысит размер программы и снизит скорость, но за стек можно быть спокойным.

"Line numbers". *Номера строк.* Когда этот переключатель находится в положении "On", компилятор помещает в объектный файл информацию о соответствии номеров строк исходного текста модуля участкам кода программы. Линкер помещает эту информацию в EXE-файл. В итоге вы имеете возможность отлаживать программу каким-нибудь отладчиком (Turbo Debugger или CodeView, например), наблюдая процесс выполнения программы по строкам исходного текста, а не по машинным командам. Важно отметить, что такая возможность дается почти задаром - хотя размеры OBJ и EXE-файлов слегка увеличиваются, в оперативной памяти программа занимает ровно столько же места, как и в режиме "Line numbers...Off". Происходит это потому, что информация о номерах строк, как и вся другая отладочная информация, помещается в EXE-файле отдельно и при "штатном" запуске программы (из ДОС) в оперативную память не загружается. Если же программу запускает отладчик, он считывает эту информацию самостоятельно. Все же для сторонников жесткой экономии дискового пространства оставлена возможность отказаться от предлагаемого сервиса и установить переключатель в положение "Off".

"OBJ debug information". *Отладочная информация в OBJ-файле.* Последняя из опций меню **"Options/Compiler/Code generation"** управляет выводом в объектный файл информации для отладчика (встроенного или отдельного): данных об используемых в модуле именах, типах переменных и т.п. Информация эта занимает в OBJ-файле весьма заметный объем. Здесь уместно сказать, что будучи выведенной в OBJ-файл (положение переключателя **"On"**), отладочная информация совсем не обязательно попадет и в EXE-файл - это зависит от значения переключателя **"Debug/Source debugging"** в момент сборки выполняемой программы. Поэтому при необходимости сгенерировать EXE-файл объемом поменьше не нужно, как это часто делается, перетранслировать все модули программы без отладочной информации, а достаточно лишь провести командой **"Compiler/Link"** новую сборку, установив упомянутый переключатель в положение **"None"**. Заметим еще, что при компиляции в режиме **"OBJ debug information...On"** не играет роли значение переключателя **"Options/Compiler/Code generation/Line numbers"**: информация о номерах строк попадает в OBJ-файл всегда. Если модуль скомпилирован без отладочной информации, отладчику будут недоступны идентификаторы объектов, используемых модулем. Возможность же установки точек прерываний и пошагового выполнения (речь сейчас идет о возможностях встроенного отладчика среды Turbo Си) зависит в этом случае от режима компиляции **"Options/Compiler/Code generation/Line numbers"**.

"Optimization". *Оптимизация.* В этом меню собраны параметры, управляющие работой по оптимизации объектного кода, проводимой компилятором.

"Optimize for". *Цель оптимизации.* Оптимизацию кода можно определить как процесс выбора из набора последовательностей команд, реализующих заданное действие, наилучшей по какому-либо показателю. Обычно показателей два - скорость программы (выбирается последовательность команд, выполняющаяся за минимальное время) и ее объем (лучшей считается самая короткая последовательность). Не всегда самая быстрая последовательность является самой короткой, и наоборот. Какую из последовательностей выбирать в случае такого конфликта, диктует компилятору значение переключателя **"Optimize for"**. Если он установлен в положение **"Size"** (это обычное его значение), выбирается самая короткая. Если же нужно максимально ускорить программу и за это не жалко заплатить длиной кода, переключатель устанавливают в **"Speed"**.

"Use register variables". *Использование регистровых переменных.* Для ускорения работы программы пользователь может указать компилятору, что те или иные локальные переменные функций должны располагаться на регистрах, что делает чтение и запись этих переменных более быстрыми. Делается это при помощи модификатора **register**, указываемого при описании переменной. Однако использование этого модификатора в программе еще ничего не значит. Будут ли локальные переменные действительно распределены на регистры, зависит от значения переключателя **"Use register variables"**. Если переключатель находится в положении **"On"**, то у каждой функции две словных локальных переменных (если, конечно, столько есть) будут размещаться на регистрах (для любителей деталей уточним - на SI и DI). Предпочтение при этом отдается переменным, в описании которых указан модификатор **register**. Среди прочих равных для распределения на регистры выбираются переменные, текстуально описанные раньше других.

Таким образом, в Турбо Си вполне можно обходиться без ключевого слова `register`, описывая самые популярные локальные (повторим, только словные!) переменные перед остальными. Если переключатель `"Use register variables"` находится в положении `"Off"`, никакие локальные переменные на регистры не попадут, вне зависимости от указания `register`. Программу это замедляет, иногда даже заметно. Тем не менее необходимость отказа от использования регистровых переменных иногда имеет место. Например, вам хочется вызвать ассемблерную подпрограмму, про которую точно известно, что она портит регистры `SI` и `DI` (или даже просто неизвестно, сохраняет ли она их значения). Без упомянутого переключателя здесь обойтись трудно - после возврата из этой подпрограммы никто не поручится за то, что значения локальных остались без изменений. Здесь необходимо заметить, что функции файла, затранслированного без регистровых переменных, могут быть вызваны только из функций, которые компилировались в этом же режиме - иначе возможны неожиданные результаты. Обратная картина - функция, не пользующаяся регистровыми переменными, вызывает функцию, их использующую, - вполне допустима.

"Register optimization". Оптимизация использования регистров. Довольно эффективный способ оптимизации, при котором генератор кода следит за тем, значения каких переменных сохраняются на регистрах, с тем, чтобы при необходимости использовать их значения читать их из не памяти, а из регистра, что гораздо быстрее и короче. К сожалению, эта оптимизация для Си не всегда корректна. Возможны ситуации, когда значение переменной в памяти изменяется неявно, компилятор этого заметить не может и использует значение, сохраняющееся на регистре и уже неверное. Здесь, конечно, не обойтись без примера, несколько искусственного, но как иллюстрация вполне подходящего.

```
#include <stdio.h>
void main (void)
{
    int a,b,*c;
    a=1;
    b=a;    /* a считывается на регистр и записывается в b, */
           /* при этом значение a осыдается на регистре, о */
           /* чем компилятору известно. */

    c=&a;
    *c=2;   /* Значение a в памяти изменяется, но компилятор */
           /* этого не знает, поэтому считает, что */
           /* на регистре сохранилось актуальное значение a */
    printf("\n%u",a);
           /* В качестве параметра printf передается значение */
           /* a, взятое с регистра, то есть 1, тогда как */
           /* настоящее значение a есть 2. */
}
```

Если этот пример не напугал вас настолько, чтобы раз и навсегда установить переключатель в положение `"Off"`, примите совет: когда присваивание через указатель (подобное оператору `*c=2` из примера) "не работает" (это можно увидеть при помощи отладчика), попробуйте отклю-

чить оптимизацию использования регистров - может помочь. Другой способ борьбы с некорректной оптимизацией использования регистров - использовать модификатор `volatile` при описании переменных, доступ к значениям которых может осуществляться более чем одним способом одновременно. Так, если в нашем примере описать а как `volatile int`, все пройдет верно - компилятор считает, что переменная `a` может изменить значение в любой момент и на регистре ее не "помнит". Нам этот способ напоминает известный метод борьбы с будущими ушибами при помощи соломы - но, возможно, кому-то он сослужит добрую службу.

"Jump optimization". Оптимизация команд переходов. Нередки случаи, когда приходится генерировать команды переходов, имеющие назначением другой переход. Примером может служить следующий оператор:

```
while (a) {
    switch (b) {
        case 1 :
            ....
            break;
        case 2 :
            ....
            break;
        default:
            ....
    } /* точка выхода из switch */
}
```

По смыслу операторов `switch`, `break` и `while` каждый из операторов `break` означает переход в точку, обозначенную комментарием. Единственное действие, которое производится в этой точке, - переход навстречу, на проверку условия продолжения цикла. Естественная и совершенно безопасная оптимизация здесь - для операторов `break` генерировать переход сразу на проверку условия цикла. Это и делается, если переключатель "Jump optimization" установлен в положение "On". Причина, по которой может понадобиться отключить эту оптимизацию, связана с отладкой.

Дело в том, что описанная оптимизация связана с определенной реорганизацией объектного кода, и может исказить связь кода с исходным текстом. В результате в некоторых ситуациях, более сложных, конечно, чем наш пример, порядок выполнения программы, показываемый отладчиком на экране, может быть неожиданным с точки зрения семантики Си. Другая возможная неприятность - отказ отладчика установить точку прерывания в строке, на первый взгляд содержащей вполне нормальные операторы (перед запуском выдается сообщение "Invalid breakpoint"). Если вас это беспокоит, перекомпилируйте программу в режиме `Jump optimization...Off`.

"Source". Небольшое подменю опций, управляющих тем, как компилятор воспринимает исходный текст.

"Identifier length". Длина идентификатора. Язык допускает использование идентификаторов любой длины, но обычно значащими считают только первые 32 символа каждого идентификатора (это означает, что любые два имени, совпадающие по первым 32-м символам, считаются одинаковыми - ужасно неприятное ограничение!). В других системах, в частности, в системе UNIX, значащими считаются 8 символов, возможно, где-то есть другие соглашения. У вас есть возможность изменить соглашение для Турбо Си, правда, только в сторону уменьшения количества значащих

символов, введя в этой позиции любое число от 1 до 32. Сделайте своему злейшему врагу 1 значащий символ и спрячьтесь. Удовольствие гарантировано!

"Nested comments". Вложенные комментарии. Вообще-то употребление вложенных комментариев служит признаком извращенного ума, но иногда хочется временно отключить какой-либо участок программы и по каким-либо причинам (невежество, лень, и т.п.) не годится использование условной компиляции (имеется в виду конструкция `#if 0... #endif`). Наиболее естественный способ сделать это - "закомментировать" этот участок. Если он в свою очередь содержит комментарии, результат такой операции не будет соответствовать стандарту Си, а "раскрывать" все внутренние комментарии - занятие малоинтересное. В этом случае помогает переключатель **"Nested comments"**. Когда он находится в положении **"On"**, компилятор обрабатывает вложенные комментарии, **"Off"** - ругается. Стандартное положение - **"Off"**.

"ANSI keywords only". Только ключевые слова стандарта ANSI. Бывают в жизни случаи, когда приходится писать программу, заранее зная, что ее придется переносить на другую машину или транслировать разными компиляторами. Конечно, при этом нужно строго соблюдать стандарт. Для Си стандартом является так называемый **"АНСИ Си"**. Турбо Си содержит ANSI Си в качестве подмножества, но упомянуть, что в Турбо Си от стандарта, а что - местное расширение, довольно сложно. Компилятор Турбо Си может оказать помощь программисту в обнаружении нестандартных конструкций. Если переключатель **"ANSI keywords only"** установлен в режим **"On"**, компилятор трактует все ключевые слова, не входящие в стандарт, как пользовательские идентификаторы. Этот режим может пригодиться не только при разработке переносимых программ, при переносе программ из какой-либо другой системы в Турбо Си. Добавим, что режим **"ANSI keywords only...On"** определяет макропеременную `__STDC__`, и этот факт достоин использования.

"Errors". Ошибки. Возможно, правильнее было бы назвать это подменю **"Предупреждения"**, так как обработке ошибок трансляции посвящена только одна из его позиций, остальные же шесть управляют выдачей предупреждающих сообщений компилятора.

"Errors: stop after...". Ошибки: после скольких останавливаться. Число, введенное в этой позиции, говорит, после скольких обнаруженных ошибок (предупреждения не в счет) останавливать трансляцию. Это число может изменяться от 0 до 255. 0 означает, что файл всегда транслируется до конца, независимо от числа ошибок.

"Warnings: stop after...". Предупреждения: после скольких останавливаться. В точности то же самое, что предыдущая позиция, но речь идет не об ошибках, а о предупреждающих сообщениях.

"Display warnings". Выводить ли предупреждения. Чрезвычайно вредный переключатель в руках неопытного пользователя. Как мы увидим ниже из подробного разбора всех предупреждений, каждое из них указывает на возможную ошибку, и нет случая, когда отношение к предупреждениям как к "чесухе, которую мелет транслятор из-за излишнего ума" взвело программиста в изнурительный поиск ошибок, на которые, будь он умнее, ему указал бы транслятор, но рот транслятора зажат установкой переключателя **"Display warnings"** в положение **"Off"**.

Наш совет, просьба, заклинание или как хотите: никогда не отключайте вывод предупреждений без крайней необходимости (должен признаться, что за всю практику не с такой необходимостью сталкиваться не приходилось)! Если же вы измучены одним-двумя надоедливými предупреждениями и не хотите избегать их из-за соображений чистоты стиля или каких-нибудь еще, отключите только их. Для отключения можно использовать два способа: меню, описанные ниже (этот не советуем: ни за что не вспомните включить предупреждение обратно!) и указания компилятору `#pragma warn` - всем хороший способ, жаль только, делающий программу непереносимой. Состоит этот способ в следующем: перед тем местом программы, которое мучает вас неоправданными, по вашему мнению, предупреждениями, пишется строка

```
#pragma warn -xxx ,
```

а после этого места

```
#pragma warn +xxx .
```

Первая из строк подавляет выдачу того предупреждения, которое обозначено в нашем примере как `xxx` (о конкретных обозначениях ниже) вплоть до особого разрешения. Вторая это разрешение дает, и предупреждение вновь может быть выдано. Если вторая строка опущена, данного предупреждения не будет до конца трансляции файла. Что касается `xxx`, это - символическое трехбуквенное имя, которое имеет каждое из предупреждений. Например, чтобы отключить предупреждение "Zero length structure", нужно написать

```
#pragma warn -zst ,
```

то есть `zst` - имя этого предупреждения. Ниже при описании конкретных предупреждений мы будем давать в скобках их символические имена, используемые в указании `#pragma warn`.

Итак, оставшиеся позиции меню "Options/Compiler/Errors" целиком заняты переключателями "On"- "Off" для каждого из 27 видов предупреждений, выдаваемых транслятором. Предупреждения сгруппированы в меню "по темам". Ниже мы рассмотрим каждое из этих подменю, при необходимости используя примеры программ.

"Portability warnings". *Предупреждения о транспортабельности.* Это меню содержит предупреждения, связанные с потенциальной нетранспортабельностью использованных в программе конструкций. Как правило, использование этих конструкций на IBM PC проблем не вызовет, однако при переносе на другую ЭВМ программа может изменить свое поведение. Часто, однако, предупреждения этой группы указывают на настоящие ошибки.

"Non-portable pointer conversion" (rpt). *Нетранспортабельное преобразование указателя.* Компилятор встретил попытку использовать указатель там, где по контексту ожидается целое, либо наоборот, целое вместо указателя. Чаще всего это просто ошибка, но иногда сознательный шаг, сделанный с надеждой на то, что компилятор преобразует тип самостоятельно. В последнем случае избавиться от предупреждения можно очень просто - обозначить преобразование типа средствами языка. Пример: следующий фрагмент программы вызовет предупреждение о нетранспортабельном преобразовании:

```
void func (void far *int_ptr);
/* функция имеет параметр - указатель */
void call (void)
```

```
{
    int int3=3;
    func(int3*4); /* в качестве аргумента - целое значение */
}
```

Выход, конечно, в том, чтобы записать вызов функции с преобразованием параметра, например, так:

```
func((void far*)(int3*4))
```

Отметим, что константа 0 преобразуется к любому ссылочному типу без предупреждений и обозначает ссылку "ни на что" (аккуратные люди все же используют в качестве обозначения "пустого" указателя стандартную константу NULL).

"Non-portable pointer assignment" (apt). *Нетранспортабельное присваивание указателя.* Оператор присваивания содержит в левой части указатель, а в правой - выражение какого-то типа, ссылочным не являющегося. Или же наоборот, слева от знака присваивания - не указатель, справа - указатель. Если это не ошибка, используйте преобразование типа, как и в предыдущем примере. Здесь тоже особый случай - присваивание указателю нуля, допускаемое свободно.

"Non-portable pointer comparison" (cpt). *Нетранспортабельное сравнение указателей.* Третий случай опасного обращения с указателем, родственный с предыдущими двумя. На этот раз указатель сравнивается с значением не ссылочного типа, отличным от константы 0. Метод борьбы с этим предупреждением точно тот же, что и с двумя предыдущими.

"Constant out of range in comparison" (rng). *Константа вне диапазона значений при сравнении.* Вы пытаетесь сравнить значение целого типа с константой, лежащей вне множества значений этого типа, например, сравнивать переменную типа char с константой 256 не имеет смысла - значения типа char бывают только от -128 до 127 (a unsigned char - от 0 до 255). Часто такое предупреждение указывает на то, что вы забыли при описании переменной задать модификатор unsigned.

"Constant is long" (cln). *Константа имеет тип long.* Компилятор встретил десятичную константу, большую, чем 32767 (максимальное значение типа int), либо шестнадцатеричную (или даже восьмеричную) константу, задающую значение, не уместящееся в слове (больше, чем 65535). В этом случае считается, что константа имеет тип long и занимает 2 слова. Чтобы избавиться от предупреждения, проще всего явно обозначить длину константы, приписав к ней l или L, например 65536 l. Часто такое предупреждение возникает, когда пытаются сравнивать значения типа unsigned int с целыми константами, уместящимися в слове, но большими, чем 32767. Такие константы, как уже было сказано, считаются длинными, сравнение, впрочем, выполняется правильно. Чтобы "превратить" константу в словную и заодно избавиться от предупреждения, нужно обозначить преобразование к типу unsigned int (например, (unsigned) 32768), либо приписать к константе u или U (32768U).

"Conversion may lose significant digits" (sig). *Преобразование может привести к потере значащих цифр.* Предупреждение всегда выдается при автоматическом преобразовании от типа long int (или unsigned long) к типу int (или unsigned int). Если описания компилятора необоснованны и вас интересует только младшее слово длинного целого, предупреждение можно подавить явным обозначением преобразования типа к целому.

"Mixing pointers to signed and unsigned char" (ucsp). Смешение указателей на типы signed char и unsigned char. Требуется преобразование от типа signed char* к типу unsigned char* или наоборот. Вещь вполне безопасная, но на других машинах может привести к неприятностям. Как и обычно, во избежание предупреждения лучше всего преобразовывать типы явно.

"ANSI violations". Нарушения стандарта ANSI. Это меню включает предупреждения о использовании ошибочных с точки зрения стандарта ANSI конструкций, тем не менее осмысленных в Турбо Си. Большинство из них чаще всего указывают на ошибки программирования, хотя можно вообразить ситуации, когда это не так. Эти предупреждения родственны предупреждениям предыдущей группы - предупреждениям о нетранспортабельности, так как при трансляции программы другим Си-компилятором, скорее всего, они превратятся в ошибки.

"XXXXXXXXX" is not part of structure" (str). Идентификатор - не поле структуры. Чтобы стало понятно, о чем речь, приведем пример.

```
/* структура для работы с машинным словом двумя способами: */
union two_bytes_word {
    unsigned int word; /* двухбайтовое целое */
    struct {           /* или два отдельных байта */
        unsigned short low_byte, high_byte;
    } bytes;
} w, *word_ptr;
void main (void)
{
    w.word=0;
    word_ptr=&w;
    word_ptr->low_byte=1; /* обращаемся к полю структуры */
    /* bytes, минуя один уровень вложенности структур */
}
```

В последнем операторе поле структуры bytes, которая, в свою очередь, является частью объединения two_bytes_word, используется так, как если бы это поле непосредственно входило в объединение. Компилятор Турбо Си это безобразие допускает и генерирует все правильно, но в oznaменование того, что от других компиляторов ожидать такого либерализма не стоит, выдает предупреждение. Выписывайте обращение к полю подструктуры полностью (для данного примера word_ptr->bytes.low_byte), и этих предупреждений у вас не будет, равно как не будет повода обвинить вас в нарушении принципов цивилизованного программирования.

"Zero length structure" (zst). Структура нулевой длины. Вы ухитрились описать структуру без полей. Воспользоваться этой структурой компилятор вам не даст, да и непонятно, как это можно было бы сделать.

"Void functions may not return a value" (voi). Функциям, описанным как void, не разрешается возвращать значение. В функции, описанной как не имеющая возвращаемого значения (void), встретился оператор return <выражение>. Выражение игнорируется, скорее всего, в программе допущена ошибка.

"Both return and return of a value used" (ret). В теле одной и той же функции использованы операторы return сразу в двух формах. Вполне ясное указание на ошибку, если учесть, что для функций, возвращающих значение, оператор "return;" приведет к возврату неопределенного значе-

ния, а для функций, описанных как `void`, выражение оператора `return` игнорируется (см. предыдущий абзац).

"Suspicious pointer conversion" (sus). *Подозрительное преобразование указателя.* Иногда бывает полезным преобразовать указатель так, чтобы он трактовался как ссылка на значение другого типа, нежели тот, что задан в описании указателя. Чаше всего это необходимо для сравнения разнотиповых указателей или для передачи указателей-параметров функций. Преобразование указателей на разные типы может проводиться автоматически (более того, часто такое преобразование не стоит ни одной команды объектного кода, так как указатели на любые типы представляются одинаково). Тем не менее выдается предупреждение, в иных случаях помогающее обнаружить очень неприятные ошибки, отловить которые при отладке не так просто. Если преобразование указателей входит в планы программиста, его лучше обозначить явно. Кроме того, не карается предупреждением преобразование указателей, один из которых ссылается на тип `void`.

"Undefined structure 'XXXXXXXX'" (stu). *Неопределенная структура.* Языком не запрещается описывать указатели на структуры, которые еще не объявлялись. Ожидается, что сама структура будет описана ниже. Если же описания этой структуры до конца файла не встретилось, выдается предупреждение. Это предупреждение, как правило, не ходит в одиночку, а сопровождается сообщениями об ошибках при попытке работать с неизвестной структурой через описанный указатель.

"Redefinition of 'XXXXXXXX' is not identical" (dup). *Неэквивалентное переопределение макроса.* В тексте встретилась инструкция препроцессора `#define`, определяющая макропеременную, уже определенную в данном файле, причем новое определение не совпадает со старым. Нужда в переопределении иногда возникает, и чтобы компилятор не выдавал предупреждений, лучше всего перед новым определением уничтожить старое командой `#undef` <имя макропеременной>.

"Hexadecimal or octal constant too large" (big). *Слишком большая шестнадцатеричная или восьмеричная константа.* Предупреждение связано с использованием в строках или символьных константах задания символа с использованием его шестнадцатеричного или восьмеричного кода. Символов с кодами большими, чем 255, не существует, так что предупреждение выдается, если в качестве кода использована константа, в десятичном выражении задающая число более 255, например, `'\477'` или `"\x35cents"`. Второй пример иллюстрирует тот факт, что при обработке символов, заданных шестнадцатеричными или восьмеричными кодами, кодом считается максимально длинная последовательность цифр. Поэтому на первый взгляд невинная запись строки `"5cents"`, в которой символ `'5'` задан шестнадцатеричным кодом, приводит к предупреждению, а заодно и к ошибке `"Numeric constant too large"` (*"Числовая константа слишком велика"*). Ведь символ `'c'` компилятор считает продолжением записи шестнадцатеричного числа, начинающегося с `0x35`. В качестве выхода можно предложить разделение кода и следующих за ним символов, например, пробелом: `"\x35 cents"`.

"Common errors". *Обычные ошибки.* В этом меню собраны предупреждения, зачастую указывающие на стандартные при программировании на Турбо Си ошибки. "Ошибки" эти не делают программу синтаксически неправильной, более того, могут быть "допущены" сознательно (как правило, это случается в недооплаченных или недоотлаженных местах программы). Тем не менее, как объект охоты, их не пожелаешь никому. Добавим к

сказанному, что для того, чтобы выдавать предупреждения этой группы, от компилятора требуется значительный (по крайней мере, для его скорости) интеллект - тем сильнее резон никогда их не отключать, хотя бы из уважения к усилиям разработчиков.

"Function should return a value" (rvl). *Функции следовало бы вернуть значение.* В функции, объявленной как возвращающая, значение (не void и не int, о функциях типа int чуть ниже), использован оператор return без выражения. Результат, который будет возвращен при использовании этого оператора, не определен, так что это чаще всего ошибка. Для функций типа int такое предупреждение не выдается. Объясняется это тем, что в старых версиях Си не было способа описать функцию, не возвращающую значение (ключевое слово void появилось позже), и эту роль принято было отдавать целым функциям. Все же и в функциях типа int это предупреждение может быть выдано, если последний перед закрывающей тело фигурной скобкой оператор - не return.

"Unreachable code" (rch). *Недостижимый код.* Вслед за оператором перехода (break, continue, goto, return) нет ни метки, ни закрывающей операторной скобки. Это означает, что на следующий оператор управление никогда не будет передано. Кроме того, компилятор пытается выявить такие циклы (while и for), у которых тело никогда не будет выполняться, вроде

или

```
for (i=1; i = 10; i++) {...} ,
```

и для них выдается то же самое предупреждение. Если программа содержит цикл, из которого нет выхода (условие окончания никогда не выполнится, как в

```
while (1) {...} ,
```

и тело цикла не содержит операторов break), следующий за ним оператор также может быть недостижимым.

"Code has no effect" (eff). *Код не имеет эффекта.* Обнаружен непустой оператор, выполнение которого не меняет значений никаких переменных, например:

```
a+b;
```

или

```
if (a == 1);
```

Чаще всего это предупреждение указывает на ошибку из разряда опечаток.

"Possible use of 'XXXXXXXX' before definition" (def). *Возможное использование переменной до присваивания ей значения.* Использование неинициализированных переменных - ошибка довольно неприятная, тем больше причин радоваться способности компилятора их обнаруживать. Делает он это очень просто: если переменная встретилась в выражении текстуально раньше, чем она же была использована в левой части оператора присваивания, то предупреждение и выдается. Присваиванием здесь считается и передача адреса переменной в качестве параметра вызова функции. Бывают (впрочем, нечасто) ситуации, когда компилятор оказывается слишком умным и обнаруживает использование неинициализированной переменной там, где этого нет. Простенький пример:

```
void main (void)
{
```

```

int i,j;
for (j=1;j <= 100;j++) {
    if (j != 1)
        i+=j; /* здесь будет предупреждение */
    else
        i=0; /* присваивание работает раньше, чем */
              /* используется значение i */
}
}

```

У отдельных неуравновешенных персон в такие моменты рука сама тянется к меню с тем, чтобы отключить это предупреждение, а с ним, возможно, еще кое-какие. Сие неразумно. Если вам неохота переставлять текст программы (так, в нашем примере достаточно поменять местами ветви условного оператора, соответственно изменив условие), инициализируйте все свои локальные переменные прямо при описании, и будете застрахованы от предупреждения, а заодно и от ошибок, им обозначаемых.

"XXXXXXXXX" is assigned a value which is never used" (aus). Переменной присвоено значение, которое не используется. Функция содержит оператор присваивания указанной локальной переменной, но не содержит обращений к этой переменной в выражениях. Обычно это предупреждение наводит на размышления о правильности программы. Иногда, впрочем, все в порядке - это случай, когда присваивание использовано лишь ради побочного эффекта выражения в правой части оператора: например, при вызове функций работы с файлами часто пишут что-то вроде `i=close(f)`, а значение переменной `i`, содержащей код результата обычно безотказной функции `close`, не очень-то интересно. В этом случае можно посоветовать ничему не присваивать значения таких выражений, благо Си позволяет написать выражение-оператор `close(f)`, несмотря на то, что функция что-то возвращает.

"Parameter 'XXXXXXXXX' is never used" (par). Параметр функции нигде внутри нее не используется. Чаще всего такое предупреждение указывает на незавершенность программы. Бывает, впрочем, нередко, что ненужные формальные параметры появляются у функции исторически, например, когда для совместимости со старой версией какого-либо пакета желательно сохранить старый интерфейс. В этом случае с предупреждением приходится смириться или, на худой конец, сделать с неиспользуемым параметром какое-либо действие "для отвода глаз", например, куда-нибудь его присвоить. Бывают и случаи, когда это предупреждение указывает на серьезную ошибку, а именно: в функции описана локальная переменная, совпадающая по имени с одним из параметров. Имя локальной "накроет" имя параметра, и тот станет недоступным. Соответственно все действия над параметром, запрограммированные в теле функции, будут произведены не над ним, а над этой злополучной локальной переменной. Если такую ошибку не заметить сразу, последствия могут быть неприятными.

"Possibly incorrect assignment" (pia). Возможно, неправильное присваивание. Обычная ошибка для новичков в Си - записывать сравнение как единственный знак равенства, например:

```
if (a=b) ...
```

Компилятор Турбо Си, к счастью для них, выдает предупреждение, встретив такой оператор в качестве выражения в условиях операторов `if`, `while` и `do-while`. Заодно нейтрализуется возможность неприятной из-за своей незаметности опечатки. Однако многие любители использовать ско-

роговорочный потенциал языка Си просто обожают заталкивать в условия упомянутых операторов все, что только возможно, в том числе и операторы присваивания (простите, выражения присваивания). Понятно поэтому обада программиста, которого подозревают в ошибке, хотя он сказал именно то, что хотел сказать: переслать значение *b* в *a* и выполнять тело условного оператора, если *b* (а вместе с ним и значение выражения-условия) отлично от нуля (см. пример). Здесь разумно пожертвовать компактностью текста в пользу удобочитаемости и переписать оператор чуть длиннее:

```
if ((a=b) != 0) ...
```

Эффект этого оператора - тот же самый, но *net* предупреждения и смысл условия понятен с первого взгляда.

Наконец, последняя составляющая меню "Options/Compiler/ Errors", на которой мы и завершим затянувшийся разговор о работе с предупреждениями транслятора Турбо Си, - это меню "Less common errors", или "Менее обычные ошибки". Несмотря на название, меню не содержит предупреждений о каких-то экзотических промахах в программировании, все эти ошибки встречаются достаточно часто, хотя и пореже, чем обитатели только что рассмотренного меню "Обычные ошибки". Пренебрегая риском выглядеть занудой, повторю еще раз: приведите все позиции меню в положение "On" и больше там ничего не меняйте - выиграете от этого больше, чем проиграете.

"Superfluous & with function or array" (amp). *Избыточная операция & над функцией или массивом.* Признак неуверенного владения языком Си - употребление операции & для получения адреса функции или массива. Если не желаете сталкиваться с этим предупреждением, запомните: имя функции без круглых скобок вслед за ним, равно как и имя массива без квадратных, - уже есть адрес функции (массива). Применение к нему операции получения адреса & не имеет смысла и вполне справедливо бракуется компилятором.

"'XXXXXXXX' is declared but never used" (use). *Идентификатор описан, но нигде не использован.* Идентификатор - будь то локальная (static) функция, статическая для модуля переменная (описанная вне функций и с модификатором static) или локальная переменная какого-либо блока - описан, но на него нет ни одной ссылки в области его видимости. Чаще всего такое сообщение указывает на объекты, которые заготавливались впрок, но так и не понадобились или стали ненужными в процессе отладки. Конечно, за собой нужно убирать, это относится и к программированию.

"Ambiguous operators need parentheses" (amb). *Двусмысленное приращение операций требует скобок.* Предупреждение это выдается всегда, когда компилятор встречает две операции сдвига, сравнения или побитовой арифметики без явного обозначения порядка выполнения скобками (вариант - сложение или вычитание в соседстве со сдвигом). Для иллюстрации смысла предупреждения приведем несколько примеров. Прикиньте, в каком порядке будут вычисляться значения следующих выражений:

```
j = 1 <= i
i ^ j & i
j >> 1 + i
```

Если не заглядывать в описания Си, то не ошибиться почти невозможно. И, конечно, гораздо проще обозначить нужный порядок выполнения скобками, чем морщить лоб в попытках сообразить, чей же приоритет выше - операции сдвига или сравнения. Напоследок, чтобы показать некоторую, как бы помягче выразиться, пониженную естественность принятых в Си соглашений о старшинстве операций, расставим скобки в наших примерах, сохранив порядок вычислений:

```
(j == 1) <= i
i ^ (j & i)
j >> (1 + i)
```

"Structure passed by value" (stv). Структура передается по значению. В языке существует два способа передачи параметров функциям - по ссылке (передается адрес аргумента) и по значению (копируется и передается само значение аргумента, при этом все изменения, внесенные в него вызываемой функцией, для исходного аргумента проходят бесследно). По ссылке можно передать любое значение. Для значений, не являющихся указателями, массивами или функциями, при передаче по ссылке обычно используется операция получения адреса &. По значению передаются любые объекты, кроме массивов и функций.

В число объектов, могущих быть переданными по значению, входят и структуры. Практика программирования на Си, однако, такова, что хорошим тоном считается передача структур только по ссылке. Возможно, объясняется это тем, что передача по значению для структур обычно менее эффективна. Можно, поднапрягшись, выдумать и другие объяснения, столь же убедительные, но как бы то ни было - по значению структуры передают редко. Именно с этим и связано предупреждение о факте передачи структуры по значению. Имеется в виду, что программист, вероятно, просто забыл указать операцию взятия адреса структуры (сравните с житейской сценкой: "Молодой человек, Вы, наверное, забыли, что ходить в шортах безнравственно?"). Надоедливое это предупреждение выдается даже в тех случаях, когда в прототипе функции честно обозначено, что она принимает в качестве параметра структуру именно по значению. Поддайся здесь программист на уговоры компилятора и поставь перед именем структуры знак &, будет выдана ошибка - "несовпадение типов". А вот для тех, кто по каким-то причинам прототипами функций не пользуется, предупреждение это очень кстати - если передаешь параметр по значению, а принимаешь по наименованию, чудес не оберешься.

До сих пор как-то не было случая поговорить о двух стилях описания функций в Си. В связи со следующими двумя предупреждениями настала пора описать эти два стиля в общих чертах. Итак, функцию можно описывать двумя способами. Первый - "классический": в заголовке функции указываются только ее идентификатор, тип возвращаемого значения и имена параметров. Тип можно опускать, по умолчанию считается, что функция возвращает значение типа int. Типы параметров описываются вслед за заголовком функции. В других модулях, использующих данную функцию, ее можно объявить - поместить лишь заголовок, в котором указан тип функции и ее идентификатор.

Второй способ - "современный": функция объявляется при помощи так называемого прототипа. Прототип функции - ее заголовок, в котором

указаны тип функции, ее имя, а также типы и (необязательно) идентификаторы параметров. Прототип содержит всю информацию, необходимую для проверки в процессе трансляции корректности всех обращений к функции - использования возвращаемого значения, совпадения типов параметров и их общего числа. Описание функции должно соответствовать прототипу и само способно служить прототипом для последующих обращений к ней в этом же модуле.

Сравнение этих двух способов показывает, что современный, без сомнения, надежнее - большое число ошибок использования функции может быть выявлено еще на стадии компиляции. Классический способ на первый взгляд гибче - можно в зависимости от обстоятельств задавать на одном и том же месте параметры разных типов, менять число параметров и т.п. Гибкость эта, однако, дается чрезвычайно дорого, так как отлаживать программы, написанные в этом стиле, гораздо труднее. Все те же ухищрения, использующие классическую схему описания функций, можно провести и в современной схеме, но с гораздо большей надежностью.

Компилятор Турбо Си поддерживает обе схемы описания функций и любые их сочетания. Обычным считается использование прототипов, поэтому в случае появления в тексте программы обращения к функции, прототип которой в данном модуле не появлялся, выдается предупреждение "Call to function with no prototype" (*Вызов функции без прототипа*). При этом безразлично, содержит ли модуль объявление функции в классическом смысле. Если по каким-то причинам программа использует классическую схему, можно отключить выдачу этого предупреждения, установив соответствующий переключатель меню "Options/Compiler/Errors/Less common errors" в положение "Off" или использовав директиву компилятора

```
#pragma warn -pro
```

В этом случае отсутствие прототипа будет считаться нормальным, но если отсутствует и объявление в классическом смысле (только тип и название функции), будет выдаваться предупреждение "No declaration for function 'XXXXXXXX'" (*Нет объявления функции*), в указаниях `#pragma warn` оно обозначается как `nod`. Как уже говорилось, необъявленные функции, а также те, в объявлениях или прототипах которых не указан тип возвращаемого значения, считаются возвращающими тип `int`.

В самом конце меню "Options/Compiler" имеется позиция, назначение которой большинству пользователей уж точно непонятно. Официальное ее объяснение "Позволяет изменять имена сегментов, групп и классов, выбираемые по умолчанию для секций Code, Data и BSS", как правило, картины не проясняет. Не будем вдаваться в подробности и мы, отделавшись общими фразами о том, что ковыряться в этих меню дозволено только крутым людям, хорошо знакомым с ассемблером и пытающимся сопрячь программы на Турбо Си с программами на каких-то других языках, причем делающим это с полным пониманием происходящего и с ответственностью.

4. СБОРКА ВЫПОЛНИМОЙ ПРОГРАММЫ

4.1. ЛИНКЕР И ЕГО "ОБЯЗАННОСТИ"

Итак, программа ваша набрана, вы пробились сквозь частокол ошибок трансляции и неспроста посмотреть, а что же у вас получилось? Для этого вы обычно нажимаете кнопку F9 (Make) или даже Ctrl-F9 (Run). На экране появляется окно с заголовком "Linking", в котором что-то происходит. Если у вас достаточно медленная машина, достаточно большая программа или достаточно быстрый глаз, вы увидите, что это "что-то" происходит в два прохода (надписи "Pass 1" и "Pass 2"). Если случилось так, что эта деятельность закончилась успешно, на диске появляется EXE-файл, заключающий в себе плоды ваших трудов в готовом к исполнению виде. А что, если нет? В окне "Messages" появились сообщения о том, что линкеру чего-то мало, что-то он не находит, или вообще нечто невразумительное вроде "Fixup overflow". Во многих случаях это означает, что предчувствие близкой отладки вас обмануло, и для ее запуска придется измного (или немало - зависит от ситуации) поработать. Что делать? Этот вечно актуальный вопрос мы и рассматриваем в данной главе.

Для начала вернемся к вопросу о том, зачем нужен линкер и что требуется ему для спокойной работы. Компилятор изготавливает файлы с расширением OBJ - по одному для каждого из модулей программы (имя модуля становится и именем OBJ-файла). Каждый OBJ-файл, помимо машинного кода для операторной части программы, содержит список глобальных объектов - как определенных в этом модуле, так и определенных неизвестно где, но модулю нужных. Последние представлены в коде условными "именами", причем в разных модулях "имена" для одного и того же объекта вполне могут различаться. Задача линкера (сознательно отвлечемся от некоторых деталей) - собрать все OBJ-файлы в один EXE-файл и заменить в коде все "имена" внешних объектов на их действительные месторасположения в программе. Именно этот процесс, называемый редактированием внешних связей, превращает набор заготовок - OBJ-файлов - в выполняемую программу.

В этой упрощенной картине не упомянуто важное понятие - библиотеки объектных программ. Обычно OBJ-файл даже самой простенькой программы содержит внешние ссылки на библиотечные, или стандартные функции Турбо Си. Даже если ваша программа не обращается ни к одной библиотечной функции, линкер должен к ней приделывать код, выполняющий действия по входу в программу, выходу из нее, обработке аварийных ситуаций и так далее. И этот код, и коды стандартных функций хранятся на диске в виде библиотек. Каждая библиотека - файл (в названии, как правило, расширение .LIB), состоящий из разделов, каждый из которых по

существу представляет собой обычный OBJ-файл. Раздел обычно соответствует небольшой группе тесно связанных между собой функций. Как и в каждом OBJ-файле, в разделе библиотеки записаны потребные ему внешние имена. Эти имена могут быть определены как в OBJ-файлах, так и в других разделах этой же или других библиотек. Вполне понятно, что в процессе сборки программы один раздел может "подтянуть" другие, они, в свою очередь, потребуют еще разделы, так что в готовой программе общий объем кода библиотечных функций часто оказывается гораздо больше пользовательского кода.

Итак, в качестве входной информации линкер имеет пользовательские OBJ-файлы (в случае многомодульной программы их набор определяется файлом проекта), а также библиотеки, из которых извлекаются необходимые программе разделы. Стандартные библиотеки Турбо Си используются автоматически, если же пользователь хочет включить в дело какие-либо другие библиотеки, их имена также должны быть записаны в файле проекта.

Директории, в которых линкер ожидает встретить все требующиеся ему по ходу дела файлы (OBJ- и библиотеки), задаются в среде Турбо Си при помощи меню "Options/Directories". OBJ-файлы ищутся в директории, записанном в позиции "Output directory" (напомним, если там ничего не записано, то в текущем). Исключение составляют файлы с именами вида C0x.OBJ (x - буква, обозначающая модель памяти). В них содержатся уже упомянутые коды входа, выхода и обработки аварий в программе, эти файлы ищутся дополнительно в директории, указанном для поиска библиотек. Как видим, мы слупавили, сказав, что эти коды хранятся в библиотеках, но, согласитесь, были недалеки от истины. Библиотеки ищутся в директориях, заданных в "Library directories", причем если задано более одного директория, то поиск происходит в порядке перечисления. Это дает возможность, например, изменить одну из стандартных библиотек и использовать ее, вставив имя директория с новой версией перед директорией, где содержатся все стандартные. Если ни в одном из указанных директориев библиотека не найдена, просматривается текущий директорий (обратите внимание на не совсем обычный порядок просмотра: в текущем директории - в последнюю очередь).

4.2. ОПЦИИ ЛИНКЕРА

Работой линкера управляют опции, задаваемые в среде Турбо Си при помощи меню "Options/Linker". Посмотрим на них.

"Map file". Генерация файла с картой сборки. В файл карты сборки (название состоит из имени EXE-файла и расширения .MAP) линкер записывает отчет о проделанной работе: какие модули и в каком порядке вошли в выполняемую программу, длины кодов модулей, расположение глобальных переменных и функций и тому подобное. В распоряжении пользователя есть четыре возможности: "Off" - не генерировать файл сборки, "Segments" - генерировать, но включать в него только информацию о собранных модулях, "Publics" - включать информацию только о глобальных идентификаторах, наконец, "Detailed" - создать файл с полной информацией о сборке. Файл карты создается (конечно, если не выбрано "Off") в директории, указанном в позиции "Output directory" меню "Options/Directories", то есть там же,

где и EXE-файл. Более подробная информация о структуре файла карты сборки и способах его использования приведена в конце этой главы.

"Initialize segments". Инициализировать ли сегменты. Здесь сложно сказать что-нибудь более-менее понятное каждому читателю, не углубляясь в вопросы, лежащие за рамками этой книги. Ограничимся тем, что констатируем: простому пользователю Турбо Си скорее всего никогда не понадобится этот переключатель. И еще добавим, что непростому - тоже. Пока не понадобится, держите его в положении "Off".

"Default libraries". Библиотеки по умолчанию. Некоторые компиляторы (только не Турбо Си) записывают в OBJ-файл имена библиотек, в которых нужно искать используемые в модуле внешние имена. Предполагается, что линкер на это должным образом отреагирует. Если вы пытаетесь скомпоновать свою программу с модулями, транслированными таким компилятором, может потребоваться включить эту опцию ("On"). Обычно этого не требуется.

"Graphics library". Графическая библиотека. Говорит линкеру о том, есть ли смысл заглядывать в поисках нужных имен в графическую библиотеку Турбо Си (файл GRAPHICS.LIB). Если ваша программа не пользуется подпрограммами графической библиотеки (они описаны в файле graphics.h), есть смысл установить этот переключатель в положение "Off": сборка пойдет чуть быстрее.

"Warn duplicate symbols". Предупреждать ли о дважды определенных именах. Когда этот переключатель находится в положении "On", линкер будет выдавать предупреждения о каждом из внешних имен, определенных более чем в одном модуле. Это может быть признаком серьезной ошибки. Представьте, что вы ненамеренно завели в двух разных модулях две функции с одинаковыми именами. По принятым правилам линкер будет все обращения по этим именам относить к функции, которую он встретил первой. Если нет предупреждения, найти такую ошибку может быть непросто. Есть еще одна тонкость, которую нужно иметь в виду. Предупреждения о дублировании имен выдаются обо всех повторяющихся идентификаторах, которые содержатся в библиотеках, даже если они не используются в программе. Если вы пользуетесь нестандартными библиотеками (в стандартных совпадений нет), может случиться, что вас вдруг предупредят об идентификаторах, которые вам вовсе не интересны. Отключать из-за этого выдачу предупреждений все же не стоит.

"Stack warning". Предупреждение об отсутствии стека. Если этот переключатель находится в положении "On", при сборке программы из OBJ-файлов, скомпилированных в модели памяти tiny, линкер выдает предупреждение "No stack". Это нормально, но только в модели tiny. Если после трансляций в другой модели (она задается в меню "Options/Compiler/Model") получено такое предупреждение, стоило бы проверить, все ли модули программы затранслированы в выбранной модели. Так как нет простого способа проверить по OBJ-файлу, в какой модели затранслирован модуль, самым разумным в этой ситуации будет запустить команду Build - она даст гарантию одинаковой модели. Если же и после этого линкер выдает предупреждение, то это случай тяжелый. Можно предположить, например, что какие-то злые шутники подменили вам файл S0x.OBJ или библиотеку Sx.LIB. Проверьте на всякий случай. Возможно, дело в ассемблерных модулях, если таковые есть в вашей программе (по-

чему-то там не определен сегмент стека). Может быть, причина в библиотеках. Короче, желаю Вам в такие ситуации не попадать.

"Case-sensitive link". Сборка с различием строчных и прописных букв. Каждый, кто программировал на Си, знает, что в этом языке различие между строчными и прописными буквами существенно, и, например, `icon` - не тот же идентификатор, что `ICon`. Однако в других языках, скажем, в Паскале, все не так. Если вы хотите включить в программу OBJ-файл или раздел библиотеки, полученный при помощи компилятора с Паскаля (или с любого другого языка, трактующего одинаково буквы разных регистров), с внешними именами могут начаться трудности. Паскалевский файл ссылается на имена Си, причем представляет их одними прописными буквами. Наоборот, программы Си ссылаются на паскалевские имена в точности так, как они описаны в Си-программе (не обязательно одними прописными), и это может не совпасть с их написанием в OBJ-файле, полученном из текста на Паскале. Для преодоления этих трудностей и предназначена возможность отключения обычного для Си различия прописных и строчных букв. Если переключатель *"Case-sensitive link"* находится в положении *"Off"*, при обработке внешних идентификаторов линкер не будет делать различия между строчными и прописными буквами (это, в свою очередь, может привести к проблемам с внешними идентификаторами Си: те же `icon` и `ICon` будут теперь считаться одинаковыми). Если ваша программа не связана с OBJ-файлами других компиляторов, разумнее, конечно, держать переключатель в положении *"On"*.

Кроме только что рассмотренного меню *"Options/Linker"* работой линкера управляет еще один переключатель, а именно *"Debug /Source debugging"*. Он имеет одно из трех значений (*"On"*, *"Standalone"*, *"Off"*) и говорит линкеру, помещать ли в EXE-файл информацию для отладчика, если такая содержится в OBJ-файлах (чтобы она имелась, транслировать файл нужно с переключателем *"Options/Compiler/Code generation/OBJ debug information"* в положении *"On"*). С точки зрения линкера *"On"* и *"Standalone"* говорят об одном и том же: помещать. *"Off"*, как нетрудно догадаться, означает "не помещать". Разница между *"On"* и *"Standalone"* в том, что первая установка делает возможным использование как встроенного отладчика Турбо Си, так и любого другого отладчика, но уже в среде ДООС. Вторая оставляет возможность использования "другого" отладчика, но отладки в среде Турбо Си не получится. За это пользователь получает больший простор в оперативной памяти, что иногда немаловажно.

Теперь немного о сообщениях линкера. Их не так много, и почти всех из них мы коснулись, обсуждая опции.

"XXXXXXXXX defined in module 'YYYYYYYYY' is duplicated in module 'ZZZZZZZZZ'". Имя *'XXXXXXXXX'*, определенное в модуле *'YYYYYYYYY'*, дублировано в модуле *'ZZZZZZZZZ'*. Сообщение о повторном определении внешнего имени. Из двух или более определений имени для использования во время выполнения выбирается первое (то, которое в сообщении обозначено *'XXXXXXXXX'*).

"Undefined symbol 'XXXXXXXXX' in module 'YYYYYYYYY'". Неопределенный символ *'XXXXXXXXX'* в модуле *'YYYYYYYYY'*. Внешнее имя, на которое ссылается модуль, не определено ни в одном другом модуле или разделе библиотеки. Частая причина такой ошибки: описывая переменную как *extern*, забывают в каком-то месте программы описать переменную без *extern* (получается, что переменная "принадлежит" модулю и место ей

будет выделено в нем). Иногда такое происходит и с функциями: пользуясь прототипом, где-то все-таки нужно описать и тело функции. Другая популярная причина наличия неопределенных имен: файл, в котором они определены, не указан в файле проекта.

"No stack". Нет стека. Обычное сообщение при сборке модулей, транслированных в модели памяти `tiny` (отдельный сегмент под стек там действительно не отводится), в других случаях - неполадка, способы борьбы с которой обсуждались чуть выше - в разговоре об опции **"Stack warning"**.

"Fixup overflow in segment 'XXXXXXXXX'". Здесь трудно привести хороший перевод, но смысл таков: *линкер встретил ссылку на внешнее имя, не соответствующую определению этого имени*. Пример такого несоответствия - использование функции как переменной или наоборот. Однако в большинстве случаев выдача этого предупреждения означает, что не все модули программы (включая разделы библиотек) затранслированы в одной и той же модели памяти. Вообще говоря, собирать в один EXE-файл модули, транслированные в разных моделях памяти, не запрещено. Нужно лишь строго соблюдать правила оформления взаимных ссылок между этими модулями. Здесь эти правила приводить не будем, любопытствующих же отошлем к руководству по Turbo Си, глава **"Advanced Programming in Turbo C"**, пункт **"Mixed-Model Programming"**. Все же предупредим: смешивать модели памяти - занятие сложное и чреватое тяжелыми ошибками.

Как помнит внимательный читатель, на конец главы мы отложили разговор о файле карты сборки: зачем он может понадобиться и что в нем написано.

Карта сборки - вещь традиционная, ее можно встретить на любой ЭВМ и в любой операционной системе, где только имеется линкер или что-то подобное. По этой карте пользователь имеет возможность пользоваться на расположение своих и библиотечных модулей в итоговом файле, узнать, сколько места занимает каждый из них (по объектным файлам компиляторов это не всегда просто понять) и по какому машинному адресу искать ту или иную функцию (переменную), когда программа будет загружена в оперативную память. Кроме того, во многих системах карта сборки - незаменимый источник информации для борьбы с аварийными ситуациями: часто сообщение об ошибке времени выполнения содержит только машинный адрес команды, где она зафиксирована, в лучшем случае, с уточнением вида **"32 шага на юг от начала подпрограммы Tree"**. Добавим еще, что карту загрузки часто используют символические отладчики и различные инструменты анализа хода выполнения программы (скажем, построение профиля программы, то есть информации о том, как часто и сколь долго работали те или иные ее части).

Все или почти все из сказанного относится и к предмету нашего разговора - карте сборки, создаваемой линкером Turbo Си.

Файл карты сборки (в дальнейшем договоримся называть его просто картой) может содержать до четырех разделов:

1. Карта сегментов.
2. Подробная карта сегментов.
3. Карта внешних имен.
4. Информация о номерах строк.

Конкретный набор разделов определяется значением переключателя **"Options/Linker/Map file"**. Значению **"Segments"** соответствует карта, содержащая разделы 1 и 4, значению **"Publics"** - разделы 1,3 и 4. **"Detailed"**

- все четыре раздела. Оговоримся здесь, что раздел 4 создается, только если хотя бы один из OBJ-файлов содержит информацию о номерах строк (то есть затранслирован в режиме "Options/Compiler/Code generation / Line numbers...On").

В конце карты всегда записываются адрес входа в программу и сообщения, выданные линкером за время сборки в окно "Messages".

Рассмотрим каждый из разделов карты, сначала определив некоторые понятия, необходимые в рассмотрении.

Выполнимая программа, записываемая в EXE-файл, состоит из сегментов. Каждый сегмент - это отрезок оперативной памяти. С точки зрения пользователя карты сборки сегмент характеризуется адресом (смещение от начала программы), длиной, именем и классом. Класс сегмента характеризует размещаемую в нем информацию: сегменты класса CODE содержат коды функций, класса DATA - константы, класса BSS - память под статические переменные, STACK - стек вызовов функций (параметры, адреса возврата, локальные переменные, пространство для хранения промежуточных данных). Имя - просто идентификатор, часто отражающий предназначение сегмента (например, по имени IO_DATA можно понять, что сегмент хранит константы модуля IO). Сегмент состоит из групп. Группа - это, можно сказать, единица сборки. Если, к примеру, линкер собирает статическую память всех модулей в один сегмент (так бывает всегда, если программы транслировались в модели памяти, отличной от huge), то этот сегмент будет состоять из групп, каждая из которых соответствует статическим данным одного из модулей. Группы могут иметь имена, хотя и не обязательно, более того, в сегмент могут входить группы с одинаковыми именами.

Карта сегментов. В первом разделе карты сборки показаны все сегменты, из которых составлена программа. Для каждого сегмента приведены его начальный и конечный адреса, длина в байтах, название и класс.

Подробная карта сегментов. Здесь приведены все группы, составляющие сегменты программы. Для каждой группы указаны следующие данные:

- адрес начала группы (в форме "сегмент:смещение");
- длина группы в байтах;
- класс и имя сегмента, в который входит группа;
- имя модуля или раздела библиотеки, из которого взято содержимое группы;
- атрибуты группы. Атрибуты группы (общее название кодировки атрибутов ACBP) кодируются двузначным шестнадцатеричным числом (байт), смысл которого здесь неважен. Желющие могут справиться в описании процессора Intel 8086 (80286) или в приложении D к фирменному руководству по Турбо Си.

Карта внешних имен. Для каждого внешнего имени приводится адрес в программе, с которым оно связано. Для функций это будет адрес первой команды кода, для переменных - адрес первого байта значения. Карта внешних имен приводится в двух формах - упорядоченная по алфавиту (Publics by Name) и по возрастанию адресов (Publics by Value).

5. УПРАВЛЕНИЕ ПРОЕКТОМ

5.1. МНОГОМОДУЛЬНЫЕ ПРОГРАММЫ

В этой главе речь пойдет о том, как в Турбо Си работать с программами, состоящими более чем из одного модуля. Никто не станет спорить с тем, что многомодульность программы на Си не являет собой чего-то экстраординарного, скорее извращением можно считать большую программу, состоящую из одного файла. Методы построения многомодульных Си-программ хорошо известны и в сочетании с соответствующим инструментарием, имеющимся в Турбо Си, позволяют относительно безбедно управляться с довольно большими (в смысле количества модулей) программами. Тем не менее все еще находятся люди, считающие своим программистским долгом все запутать так, что не только посторонний, но и сами они легко теряются в хитросплетениях межмодульных связей. Но этому разговору о средствах работы с многомодульными программами в Турбо Си предпослём небольшую беседу о технологии составления таких программ. Возможно, это и не наставит упомянутую часть программистского сообщества на путь истинный, но, может быть, окажет благотворное влияние на становление тех читателей, кто, как программист, еще только формируется. Лица, считающие себя достаточно знакомыми с предметом, чтобы не тратить время на чтение очередного зевотворного текста "о хитрахах и прототипах", могут спокойно пропустить следующие несколько страниц.

Основное, о чем пойдет ниже разговор, - это организация межмодульного интерфейса. Каждый модуль, входящий в программу, предоставляет остальным модулям услуги в виде своих функций и глобальных переменных (каждый, кроме, может быть, главного, содержащего функцию `main`, но им здесь можно пренебречь). И наоборот, большинство модулей используют сервис других модулей. Отличительной особенностью языка Си (добавим, особенностью, его не красящей) является отсутствие в языке средств раздельной зависимой компиляции. Что такое "раздельная зависимая компиляция", лучше всего объяснить на примере других современных языков - Турбо Паскаля (начиная с версии 4), Модулы 2 или Ады. В этих языках для того, чтобы воспользоваться объектом, определенным в другом модуле (назовем его экспортером), в модуле пользователя (его называют импортером) достаточно указать имя экспортера и, возможно, имя импортируемого объекта (Модуля 2, Ада; если указано только имя модуля, импортируются все его глобальные объекты). Все остальное берет на себя компилятор: достает из экспортера описание нужного объекта и проверяет правильность его использования в импортере. Если описание экспортируемого объекта в какой-то момент изменилось, нетрудно автоматически найти все модули, его использующие, и перетранслировать их с учетом изменений.

Таким образом достигается высокий уровень надежности в работе с многомодульными программами. Есть, конечно, и ограничения: например, в ряде случаев запрещен циклический импорт (простейший пример: модуль

А использует переменную из модуля В, а модуль В, в свою очередь, - другую переменную из А). Тем не менее такая схема хорошо себя зарекомендовала и используется практически во всех языках программирования, созданных за последнее время.

В Си раздельная трансляция является независимой. Это значит, что в момент компиляции модуля транслятору ничего не известно о содержимом других модулей и о глобальных объектах, ими поставляемых. Поэтому транслятор должен полагаться на сведения об импортируемых функциях и переменных, сообщенные ему пользователем. Расхождение между описанием одного и того же объекта в модуле-экспортере и модуле-импортере никак не фиксируется, и ответственность за такого рода неувязки лежит на программисте (ошибки такого рода - чуть ли не самые разрушительные и трудновываливаемые). Понятно, что с такой схемой раздельной трансляции на фоне наступления языков-потомков Паскаля Си, при всей его привлекательности, вероятно, ожидала судьба остальных "старых" языков (вспомним PL/1, помимо прочих прелестей эксплуатирующий именно описанную схему).

Но выход был найден довольно быстро: независимая трансляция была наделена свойствами зависимой при помощи чисто технологического приема, очень удачно вписавшегося в стиль программирования на Си. Суть приема в том, что для каждого модуля составляется файл, содержащий описания экспортируемых объектов. Этот файл называется *header*, что по-русски означает "заголовок". В среде программистов русский перевод почему-то не утвердился, и его так и называют "хэдер" (есть еще вариант "хидер", но за такое произношение можно получить "по мозгам" от первой же встреченной учительницы английского).

Хэдер-файл при помощи препроцессора (благо, он с самого начала был предложен авторами Си как неотъемлемая часть языка, хотя это было довольно смелым шагом) вставляется в текст каждого из модулей-импортеров. Теперь для импорта объекта не нужно копировать его описание, а достаточно употребить директиву препроцессора, и описание объекта и всех его "соседей" по экспортеру попадет в текст модуля точно в том виде, в каком они экспортируются. Соответствие описаний в хэдер-файле и в самом модуле-экспортере поддерживается путем включения хэдера в текст и этого модуля. При изменении описания в хэдер-файле или в самом модуле компилятор заметит расхождение в двух описаниях одного и того же объекта, предотвратив тем самым возможные неприятности.

Что включают в хэдер-файл? Во-первых, прототипы экспортируемых функций, включающие в себя описания типа возвращаемого значения и типов (а, возможно, и имен) параметров. Во-вторых, описания глобальных переменных, определенных в модуле, с атрибутом *extern* (напомним, что переменная, описанная как *extern*, считается определенной в другом модуле, если только нет второго описания - без *extern*). В-третьих, описания типов и макроопределений, используемых для связи с модулем. Наконец, хэдер-файл может в свою очередь включать в себя другие файлы, содержащие описания, нужные для трансляции его самого: хорошим тоном считается отсутствие необходимости включать эти файлы в каждый из модулей-импортеров.

Поясним все вышесказанное на примере. Пусть имеется модуль *POOL*, реализующий функции работы с "кучей" значений типа *VALUE*: заведение нового значения, его запись, чтение и освобождение (тип *VALUE*

будем считать описанным в другом хэдер-файле под названием VAL.H). Тогда хэдер-файл для модуля POOL.H (негласный стандарт диктует называть хэдер именем модуля-родителя с расширением .H) может выглядеть так:

```

/*===== POOL.H =====*/
/* Модуль работы с кучей объектов типа VALUE */
/* В хэдер-файл включается определение типа VALUE */
#include "val.h"
/* Константы-коды возврата */
#define NO_MEMORY -1
#define TOO_MANY_HANDLES -2
#define INVALID_HANDLE -3
#define INTERNAL_ERROR -99
/* Описание типа дескриптора значения */
typedef unsigned int HANDLE;
/* Описания переменных, определенных в модуле. - */
/* все с модификатором extern! */
extern long pool_size; /* размер кучи */ extern long reclaim_value;
/* значение для вызова */
/* сборки мусора */
/* Функции работы с кучей - только прототипы */
int OpenPool (void); /* создать пустую кучу */
int ClosePool (void); /* освободить память кучи */
/* создать значение */
int AllocateValue (int size,HANDLE *descr);
int FreeValue (HANDLE descr); /* освободить память */
/* записать значение */
int WriteValue (HANDLE descr,VALUE *val);
/* считать значение */
int ReadValue (HANDLE value,VALUE *val);
/*===== конец POOL.H =====*/
А вот как будет выглядеть сам модуль POOL.C:
/*===== POOL.C =====*/
/* Хэдер-файлы, нужные для трансляции модуля */
#include alloc.h
#include .h
/* Хэдер-файл с интерфейсом модуля */
#include "pool.h"
/* Определения переменных, экспортируемых модулем: */
/* здесь без extern; можно задавать начальные */
/* значения */
long pool_size = 65536;
long reclaim_value = 65500;
/* Определения статических переменных модуля, */
/* используемых только внутри него макроопределений */
/* и функций (переменные и функции - с атрибутами */
/* static) */
static HANDLE handles[ ];
/*...*/
#define MAX_HANDLES 512
#define BAD_HANDLE(h) (handles[h] == NULL)

```



```

/* ... */
static void Reclaim (void)
{
/* ... */
}
/* ... */
/* Определения функций работы с кучей - компилятор */
/* проверит соответствие прототипам из POOL.H */
int OpenPool (void)
{
/* ... */
}
int ClosePool (void)
{
/* ... */
}
int AllocateValue (int size,HANDLE *descr)
{
/* ... */
}
int FreeValue (HANDLE descr)
{
/* ... */
}
int WriteValue (HANDLE descr,VALUE *val)
{
/* ... */
}

Int ReadValue (HANDLE value,VALUE *val)
{
/* ... */
}
/*===== конец POOL.C =====*/

```

К примеру можно добавить еще вот что. Предположим, что модуль, использующий POOL, включает кроме файла POOL.H еще и файл VAL.H - для каких-то своих нужд (знать о том, что VAL.H уже включен в POOL.H, автор этого модуля не обязан). Тогда VAL.H будет транслироваться дважды. Большой беды в этом нет, так как повторные описания языком разрешаются. Однако, если VAL.H достаточно велик, трансляция может заметно замедлиться. Чтобы избежать этого замедления, часто хэдер-файлы оформляют примерно следующим образом:

```

/* POOL_H - имя препроцессорной переменной. */
/* образованное от имени хэдер-файла */

#ifndef POOL_H

/* далее следует текст файла */
/* ... */
/* в конце файла определяем POOL_H с тем, чтобы */
/* условие #ifndef отныне не давало транслировать */

```

```
/* весь файл. */  
#define POOL_N  
/* закрывающая "скобка" для #ifndef */  
#endif
```

Как видно из примера, небольшой трюк с препроцессором заставляет компилятор пропускать текст хэдер-файла всегда, за исключением первого появления этого текста в транслируемом модуле.

Хэдер-файл, описанный таким образом, можно включать в текст сколько угодно раз, и на скорости трансляции это практически не скажется.

На этом завершим общеобразовательную часть главы и перейдем к основному предмету - средствам работы с многомодульными программами в Турбо Си. Одним из ключевых понятий этой системы программирования является файл проекта.

5.2. ФАЙЛ ПРОЕКТА

Файл проекта содержит перечень всех исходных модулей, составляющих программу, а также используемых на этапе сборки библиотек и OBJ-файлов. Здесь же задаются зависимости между файлами: под зависимостью подразумевается необходимость перетрансляции одного файла при изменении другого. Информации файла проекта достаточно, чтобы собирать выполняемую программу командой **Link**, а также автоматически поддерживать согласованность различных модулей программы при помощи команд **Make** и **Build**.

Посмотрим, как устроен файл проекта. Это обычный текстовый файл. В самом простейшем виде он может состоять из строк, каждая из которых содержит имя файла, содержащего один из модулей программы. Как обычно, расширение **.C** можно опускать. В названиях можно указывать путь, это позволяет собирать программу из файлов, расположенных в разных директориях. Файл, составленный таким способом, уже позволяет выполнять команды **Make**, **Build** и **Link**: **Make** перекомпилирует все те модули проекта, для которых нет OBJ-файлов или дата создания OBJ-файла меньше даты последнего изменения текста модуля, **Build** просто перекомпилирует все перечисленные модули (обе эти команды после успешного окончания перекompilаций запускают **Link**), а **Link** попытается из OBJ-файлов для указанных модулей и стандартных библиотек Турбо Си создать выполняемую программу - EXE-файл. Имя EXE-файлу дается по имени файла проекта. Кстати, название файла проекта совсем не обязательно должно совпадать с названием головного модуля программы: здесь у пользователя полная свобода, в названии проекта не обязательно даже использовать расширение **.PRJ**, принятое как обычай (такой же ненавязчивый, как **.C** у файлов с текстами модулей или **.H** у хэдеров). Единственное, что нужно помнить, - это то, что имя создаваемого EXE-файла всегда совпадает с именем проекта.

Описанный простой способ задания проекта вполне достаточен для работы с маленькими программами, содержащими два-три модуля.

Для программ побольше жизненной необходимостью является использование такой важной черты команды **Make**, как способность воспринимать описания зависимостей между модулями проекта и на их основании поддерживать согласованность программы. Зависимости задаются следующим образом. После имени файла с текстом модуля в скобках перечисляются все

файлы, от которых он зависит (читай: "которые он включает директивой `#include`"). Разделителем в списке файлов может служить запятая, точка с запятой или даже пробел. Здесь также можно указывать путь и опускать расширения. С - поэтому расширения .H, обычные в этом списке, приходится задавать всегда. Зависимости косвенные (A.D включает B.H, который включает D.H) также стоит помещать в этот список. Так, если вспомнить наш пример, любой модуль, использующий функции POOL.C, должен будет иметь в своем списке зависимостей файлы POOL.H и VAL.H. Теперь после изменения текста любого из файлов проекта (модуля или хедера) командой Make можно перекомпилировать все, от него зависящее, то есть содержащие измененный файл в списке зависимостей. При определении того, изменился ли файл, Make ориентируется на дату последней редакции файла. Если дата редакции исходного текста больше даты создания соответствующего OBJ-файла, модуль подлежит перекомпиляции.

Работа по ручному отражению зависимостей в файле проекта довольно утомительна, поэтому в систему включены средства, дающие возможность автоматически отслеживать зависимости между модулями (читай ниже о переключателе "Project/ Auto dependencies").

Но это еще не все, что можно сообщить о файле проекта. Уже мимоходом замечалось, что в проект можно включать OBJ-файлы. Делается это в случаях, когда часть программы пишется не на Си, а, скажем, на ассемблере (или на любом другом языке, компилятор которого создает OBJ-файлы). Имя полученного в результате ассемблирования OBJ-файла включается в файл проекта, обязательно с расширением .OBJ. Команды Make и Build такие файлы пропускают и компилировать не пытаются, а линкер включает их в выполняемую программу без лишних вопросов.

Таким же образом используются и библиотеки объектных программ: имя файла с библиотечкой (расширение .LIB) включается в проект, и линкер добавляет этот файл в список библиотек для поиска внешних имен в ходе сборки проекта. Заметим, что OBJ- и LIB-файлы не могут иметь список зависимостей (точнее говоря, задать такой список можно, но он будет игнорироваться). Однако включать эти файлы в чужие списки зависимостей можно, это будет означать, как обычно, что при изменении OBJ- или LIB-файла соответствующий модуль перекомпилируется. Смысла, впрочем, в таком использовании имен объектных и библиотечных файлов не видно.

И последнее, что можно сказать о файле проекта. В Турбо Си имеются стандартные библиотеки и стандартные же OBJ-файлы с кодом входа в программу, выхода из нее, а также обработки аварийных ситуаций - все эти файлы обозначают родовым именем start-up. И библиотек, и start-up-файлов ровно столько, сколько моделей памяти в Турбо Си, то есть 6, по одному на модель. И стандартные библиотеки, и файлы start-up участвуют в сборке программы вне зависимости от упоминания их в файле проекта. Файлы, содержащие стандартные библиотеки, называются Sx.LIB, файлы start-up - C0x.OBJ (здесь x - буква, соответствующая модели памяти: T - tiny, S - small, L - large, C - compact, M - medium, H - huge).

Если по каким-либо причинам нужно блокировать использование библиотеки и / или файла start-up в сборке программы (конечно, при этом нужно предоставить директору какие-то эквиваленты, выполняющие те же функции и определяющие те же внешние имена, что и стандартные файлы), это можно сделать, указав пользовательские файлы start-up или "стандартные" библиотеки в файле проекта. Соглашение здесь такое. Если первым

в файле проекта упомянут файл, расширение которого - .OBJ, а имя начинается с символов C0 (например, C0BEST.OBJ), именно этот файл включается в выполняемую программу, а стандартный start-up не используется. Аналогично, если в файле проекта указан (необязательно первым) файл с именем вида Sx.LIB (например, C1.LIB; отметим, что здесь имя должно содержать ровно два символа), он используется как обычная библиотека, однако стандартная для этой модели памяти библиотека в сборке не участвует. Важно помнить, что в случае такого отключения стандартной библиотеки не используется автоматически библиотека математических функций (MATHx.LIB, EMU.LIB или FP87.LIB, в зависимости от установки переключателя "Options / Compiler/Code generation /Floating point" при компиляции модулей). При необходимости все же использовать эту библиотеку нужно явно указать ее в файле проекта.

5.3. МЕНЮ УПРАВЛЕНИЯ ПРОЕКТОМ

Обратимся теперь к меню среды Турбо Си, объединяющему команды и переключатели для работы с проектом. Называется это меню, конечно же, "Project".

"Project name". *Имя файла проекта.* В этой позиции записывается имя текущего файла проекта. Если эта позиция не пуста, то все команды Make, Build, Link и Run работают с использованием указанного здесь файла вне зависимости от того, что находится в окне редактора. Имя файла проекта дается создаваемому по нему EXE-файлу, а также (если создается) файл с картой сборки (.MAP). Как и в других местах среды, позиция, будучи выбранной, предоставляет окно для записи имени файла проекта, в этом окне можно задать лишь маску имени - тогда будет предложено выбрать файл из списка подходящих.

"Break make on". *В каких случаях прерывать Make* Выполнение команды Make - процесс многоходовый, и зачастую задолго до его окончания становится ясно, что к созданию EXE-файла он не приведет. Поэтому важно заранее задать условия, при которых команда Make прерывает свою работу после компиляции очередного файла проекта. Переключатель "Break make on" содержит 4 позиции, каждая из которых задает свои условия.

"Warnings". *Предупреждения.* Прерывает работу Make, если трансляция очередного файла завершилась с выдачей предупреждающих сообщений. Требования довольно суровое, но имеет свой смысл, когда программист склонен доводить свои программы если не до блеска, то, по крайней мере, до отсутствия каких бы то ни было претензий со стороны компилятора.

"Errors". *Ошибки.* Обычная и, похоже, самая разумная установка. Make прерывается, если компилятор обнаружил ошибки в каком-либо файле проекта, предоставляя автору немедленно заняться их исправлением. Если ошибок не обнаружено (возможно, и были предупреждения), вызывается линкер.

"Fatal errors". *Фатальные ошибки.* К разряду фатальных ошибок относятся следующие несчастья (заодно уж приведем и способы борьбы с ними).

"Irreducible expression tree". *Дерево выражения не поддается обработке.* Сообщение говорит о том, что вы наткнулись на какую-то ошибку в компиляторе. Попробуйте изменить выражение, на котором он сломался

(обычно в таких случаях помогают даже самые дурацкие изменения, вроде перестановки слагаемых или добавления пары скобок).

"Register allocation failure". *Не удастся распределить регистры.* Тоже признак ошибки в компиляторе, но в отличие от предыдущего сообщения можно с большей степенью уверенности сказать, что именно произошло. Скорее всего, вы подсунули компилятору уж очень сложное выражение, превысившее его "грузоподъемность". Попробуйте сделать это выражение попроще или разбить его на несколько частей.

"Internal undefined error". *Неопознанная внутренняя ошибка.* Яснее не скажешь. Встретив такое сообщение (равно как и два предыдущих), скорее бегите (летите, плывите, звоните, пишите) в фирму Borland International. Там вам, скорее всего, будут рады (это только у нас кто-то кое-где порой встречает глухим рычанием пользователя, принесшего ошибку) и - кто знает - может быть, даже заплатят полновесным долларом.

"Bad call of in-line function". *Неверное обращение к inline-функции.* Inline-функция - это "псевдофункция": компилятор, встретив вызов такой функции, генерирует не обращение к соответствующему телу функции участку кода, а непосредственно машинные команды, осуществляющие нужное действие. Набор таких функций в Турбо Си фиксирован (пользователь подобную функцию описать не может): `__int__`, `__abs__`, `__cli__`, `__sti__`, `__outportb__`, `__inportb__`, `__emit__`. Ошибка возникает, если пользователь использует одну из этих функций, предварительно ее не описав. Описывать их, впрочем, нужды обычно нет: все они описаны в файле `DOS.H`, исключением является `__abs__`, описанная в `STDLIB.H`. Более того, имена функций (обратите внимание на двойные подчеркивания в начале и в конце каждого имени) замаскированы макросами - привычными именами `int`, `abs`, `disable`, `enable`, `outportb`, `inportb` и `emit` соответственно.

"Auto dependencies". *Автоматическая проверка зависимостей.* Когда этот переключатель находится в положении "On", Make вместо простой проверки: не "маложе" ли исходный текст соответствующего ему OBJ-файла - проводит более жесткую проверку. Он считывает из OBJ-файла информацию о датах и временах последнего (на момент компиляции) изменения всех файлов, использованных при его создании (это сам файл с исходным текстом и все включенные в него хэдер-файлы - на любую глубину). Даты эти сравниваются с теперешними, и, если даты хоть в какой-либо паре не равны, производится перекомпиляция. Эта проверка ловит неприятный случай, когда по каким-то причинам восстановлена старая версия текста (со старой датой), а OBJ-файл остался от новой, и программа стала рассогласованной, причем Make в режиме "Auto dependencies...Off" ничего не заметит.

Плата за автоматизацию (при использовании этой возможности не обязательно записывать в файл проекта зависимости модулей) такова: поскольку проверяются даты всех хэдер-файлов, время тратится и на стандартные хэдеры, и на те, вероятность изменения которых так же низка (например, H-файлы какого-нибудь инструментального пакета). Особо нетерпеливые имеют возможность автоматику отключить ("Off"), слегка ускоряя работу Make и беря на себя ответственность за корректность записанных в файле проекта зависимостей.

"Clear project". *Очистить проект.* Команда говорит о том, что начинается работа без проекта: если теперь запустить компилятор или Make, они будут пытаться построить одномодульную программу - имя

модуля берется из "Compile/Primary C file" или из окна редактора. Заодно очищается окно "Message". Того же эффекта, но без очистки "Messages", можно достичь, введя в позицию "Project name" пустую строку.

"Remove messages". Удалить сообщения. Просто чистит окно "Message".

6. ОТЛАДКА В ИНТЕГРИРОВАННОЙ СРЕДЕ.

Встроенный отладчик в нашей скудной на удобства жизни можно сравнить разве что с выездной торговлей в том виде, в каком она, наверное, была задумана: сервис сам приходит к потребителю и скромно стоит в углу, готовый по первому зову развернуть сеть услуг, на которые в обычной ситуации приходится тратить кучу сил и времени. С этой самой торговлей его роднит и наличие немалой армии обездоленных, не имеющих доступа к благам цивилизации, - пользователей не столь продвинутых систем программирования (как на РС, так и на других машинах). Отличие, правда, в том, что обделенные программисты не требуют запрещения встроенных отладчиков во имя торжества социальной справедливости.

Версия Турбо Си 2.0 отличается от Турбо Си 1.5 не очень сильно: не считая мелочей, всего-то навсего добавлен встроенный отладчик. Но в то же время разница между этими двумя версиями - как между небом и землей. Турбо Си 1.5 была всего лишь еще одной системой программирования для Си, обладающей, правда, совмещенными редактором и компилятором, но это большое удобство не может считаться чем-то экстраординарным: хороший редактор (например, MultiEdit) вполне допускает подключение компилятора, система помощи по языку - тоже не новость (вспомним Norton Guide по Си). Один лишь штрих - отладчик, которым можно пользоваться, не выходя из среды редактора/компилятора, - поднял эту систему на высоту неописуемую. Добавим, что отладчик сей - не хилая пристройка, позволяющая устанавливать точки прерываний в машинных командах и обозревать память по шестнадцатеричным адресам, но полноценный инструмент отладки, работающий с исходным текстом на Си и имеющий множество очень приятных возможностей. Обзору этих возможностей посвящена нижеследующая глава. В ней мы сначала поговорим о том, какие действия требуются от пользователя, чтобы отладчик мог полноценно работать, затем посмотрим на набор команд, имеющийся в распоряжении программиста при отладке, разберемся со способами трассировки значений (Watch) и вычисления выражений (Evaluate).

Если вы желаете воспользоваться встроенным отладчиком Турбо Си, программу нужно к этому подготовить. Во-первых, отлаживать можно лишь те модули программы, которые были затранслированы с переключателем "Options/Compiler/Code generation/OBJ debug info" в положении "On". Мало того, в момент сборки выполняемой программы переключатель "Debug/Source debugging" должен находиться в положении "On". Что произойдет, если одно или оба из этих условий не выполнены? Если для каких-то модулей не выполнено только первое, вы не сможете следить за значениями локальных для этих модулей переменных в окне "Watch" и использовать их в вычислениях, проводимых командой "Evaluate" (стандартным ответом отладчика на попытки это сделать будет "No type information").

- "Нет информации о типе"). Установка точек прерываний и пошаговое выполнение при этом будут возможны в зависимости от установки-переключателя "Options/Compiler/Code generation/Line numbers" - только для модулей, в момент трансляции которых этот переключатель находился в положении "On". Если не выполнено второе условие, отладчиком просто не воспользоваться: при попытке запуска программы будет выдано сообщение "No debug info. Run anyway?" ("Нет отладочной информации. Все равно запускать?"). В зависимости от ответа пользователя программа будет все-таки запущена (ответ "Y") или нет.

Теперь о командах отладчика. Они собраны в трех меню верхнего уровня - "Run", "Debug" и "Break/watch". Многие из них имеют альтернативную форму вызова - так называемые горячие клавиши, будем указывать их в скобках после названия команды.

"Run". Меню содержит команды, управляющие ходом выполнения программы.

"Run" (Ctrl-F9). *Запустить*. Если программа к запуску не готова, будут вызваны компилятор и линкер. Как только выполнимая программа создана (в случае ошибок при компиляции или сборке, конечно, выполнения не будет), она запускается. Остановка произойдет либо по окончании работы программы, либо по достижении одной из заранее заданных точек прерываний, либо в результате нажатия пользователем Ctrl-Break. Пару слов о Ctrl-Break: одновременное нажатие этих клавиш не всегда сразу приводит к остановке программы. Если управление в момент нажатия находится в модуле, оттранслированном с информацией о номерах строк ("Options / Compiler/Code generation/Line numbers...On"), среда действительно прервет выполнение программы и покажет в окне редактора место, где прерывание произошло. Программу в этом случае можно запустить дальше с этого же места или начать выполнять пошагово - словом, отладку можно продолжить. Если же модуль компилировался в режиме строгой экономии (это обычная причина для отключения информации о номерах строк), на первое нажатие Ctrl-Break реакции не будет, а если нажать еще раз, то выполнение прервется и среда выдаст сообщение "User break, program terminated" - "Пользовательское прерывание. Программа завершена". После этого возобновить работу программы уже нельзя - можно только запустить ее с самого начала. Бывает, что не помогает и двойное нажатие Ctrl-Break. Означает это только одно - вы доигрались до того, что программа ваша "повесила" машину. В зависимости от тяжести случая придется либо нажать Ctrl-AltDel, либо выключить аппарат (более гуманно, конечно, надавить Reset - там, где эта кнопка есть).

"Program reset" (Ctrl-F2). *Приведение программы в стартовое состояние*. Команда предназначена на случай, когда в процессе отладки приходится запускать программу сначала, не дав ей проработать до конца. При этом сохраняются текущее содержимое окна "Watch" и установленные точки прерываний. Из памяти удаляются кое-какие данные, используемые отладчиком по ходу выполнения программы. Отсюда еще одно применение команды "Program reset": если не хватает памяти, например, на компиляцию особо большого файла, нажатие Ctrl-F2 может помочь.

"Go to cursor" (F4). *Идти до курсора*. Программа запускается (возможен вызов компилятора и линкера) и выполняется до достижения строки программы, на которой был расположен курсор в момент выдачи команды. Остановка не обязательно произойдет именно на этой строке - вполне

возможно, что раньше управление достигнет одной из ранее установленных точек прерывания. Команда F4 аналогична установке "одноразовой" (отменяемой по достижении) точки прерывания.

"Trace into" (F7). *Трассировать вызываемые функции.* Это команда пошагового выполнения, "спускающаяся" в каждую функцию, которая вызывается по ходу выполнения программы. Конечно, если вызываемая функция недоступна отладчику (транслирована не в том режиме), спуска не будет, и нажатие F7 вызовет тот же эффект, что команда простого пошагового выполнения - F8 (см. ниже). Эти две команды эквивалентны и в случае, если очередная строка программы не содержит вызовов функций.

"Step over" (F8). *Шагнуть поверх.* Действие очень похоже на F7. Разница в том, что F8 никогда не спускается в вызываемую функцию (если, конечно, ее вызов не приведет к точке прерывания). Обычно эту команду используют для отладки "отдельно взятой" функции, просто выполняя ее операторы по одному.

"User screen" (Alt-F5). *Пользовательский экран.* Все, что выводит программа в файл stdout, при выполнении в среде Турбо Си поступает на экран (напомним, переназначение стандартного вывода в среде недоступно). Обычно экран этот от пользователя скрыт, для его просмотра используется команда Alt-F5. Кроме данных, выведенных программой, на пользовательском экране можно увидеть остатки сообщений DOS, выданных перед входом в среду или во время работы в DOS при помощи команды "File/OS shell". Для возврата к экрану среды достаточно нажать любую клавишу. Заметим, что в случае, когда переключатель "Debug/Display swapping" находится в положении "None", вывод программы идет прямо на экран среды, а пользовательский экран остается без изменений.

"Debug". В этом меню - различные команды, используемые при отладке.

"Evaluate" (Ctrl-F4). *Вычислить.* Команда позволяет вычислять выражения, просматривать и изменять значения переменных. Полезна как при отладке, так и сама по себе - как неплохой калькулятор, который всегда под рукой. Подробный разговор об использовании команды "Evaluate" пойдет в конце этой главы.

"Call stack" (Ctrl-F3). *Стек вызовов.* В процессе отладки иногда бывает полезно узнать, из какого места вызвана функция, в которой сейчас находится управление. Турбо Си предоставляет такую возможность, по команде Ctrl-F3 выдавая на экран окно, в котором перечислены все активные на данный момент функции, начиная с функции main и через все ей вызванные до текущей. Список этот устроен подобно меню, и выбор из него какой-либо функции приводит к выводу в окно редактора текста этой функции, причем курсор будет расположен на строке, которая содержит вызов предыдущей в списке функции. Команда показывает только функции, скомпилированные в режиме "Options/Compiler/Code generation /Standard stack frame...On" (а также всегда функцию, в которой находится управление).

"Find function". *Найти функцию.* Отлаживая достаточно большую программу, не всегда просто сообразить, где расположена та или иная функция. Здесь на помощь приходит команда "Find function". В ответ на задание идентификатора функции она выводит в окно редактора файл, содержащий тело функции, причем располагает курсор на ее заголовке. Для того, чтобы эта команда сработала, нужно выполнить два условия. Во-пер-

вых, должен быть доступен файл, содержащий исходный текст функции (так, стандартные функции искать этой командой бессмысленно). Во-вторых, этот файл должен быть оттранслирован с информацией о номерах строк ("Options /Compiler/Code generation/Line numbers...On").

"Refresh display". *Освежить экран.* Как уже говорилось мимоходом, если переключатель **"Debug/Display swapping"** находится в положении **"None"** (или **"Smart"**, и вы упражняетесь с прямой записью в видеопамять), вывод программы на экран смешивается с картинкой, нарисованной средой Турбо Си. Несколько таких выводов - и работать в среде становится весьма затруднительно. Эта команда позволяет перерисовать весь экран Турбо Си заново. Старый, "испорченный" экран, конечно, забывается, так что перед вызовом команды перепишите на листочек наиболее ценные байты из числа выведенных программой.

"Display swapping". *Переключение экрана.* Уже не раз говорилось о том, что данные, которые программа записывает в файл `stdin`, выводятся на экран дисплея. В связи с тем, что на одном "физическом" экране среда Турбо Си вынуждена поддерживать одновременно два "логических" экрана - пользовательский и собственно экран среды, приходится предпринимать действия по переключению с одного логического экрана на другой. Переключение это обычно производится при выполнении команд запуска - **"Run"**, **"Go to cursor"**, **"Step over"** и **"Trace into"**. Когда программа запускается, дисплей переключается на пользовательский экран, так что на нем можно увидеть весь вывод, который делает выполняющаяся программа. Как только программа останавливается (закончив выполнение или достигнув точки прерывания), пользовательский экран запоминается (его можно, как мы уже знаем, посмотреть, нажав `Alt-F5`) и выводится экран среды. Переключатель **"Display swapping"** устанавливает режим, определяющий, когда нужно производить переключение экрана, а когда нет.

"None". *Никогда.* Экран среды и пользовательский экран совмещены, и никакого переключения между ними не производится. Такая возможность бывает полезна при отладке участков программы, не выводящих ничего на экран. Дело в том, что переключение экрана со среды на пользовательский приятным для глаз никак не назовешь, а в других режимах оно производится довольно часто - по разу на вызов функции или даже на выполнение оператора. Если на пользовательский экран смотреть не нужно, то, конечно, это переключение лучше отключить.

"Smart". *По-умному.* Как уже сказано, постоянное переключение экрана давит на глаза. Режим **"Smart"** - попытка облегчить ситуацию. При пошаговом выполнении среда не переключает экран, если уверена, что выполняемый оператор не повлечет вывода на экран. Определяется это довольно просто: отладчик считает, что вызов любой функции может привести к выводу, а любые другие операторы - нет. Такая простая политика оказывается довольно эффективной, и режим **"Smart"** можно порекомендовать к использованию в большинстве случаев.

"Always". *Всегда.* Экраны переключаются при любой команде запуска программы. Спрашивается, когда возникает потребность в таком режиме, если есть режим **"Smart"**, переключающий экраны только когда это нужно? В ответ можно привести случай отладки программы, производящей запись непосредственно в видеопамять, причем не через вызов функций, а простыми присваиваниями или, скажем, ассемблерными командами. Режиму **"Smart"** эти присваивания и команды кажутся вполне безопасными, тем не

меее на экран они что-то выводят, и при пошаговом выполнении попадает это "что-то" не на пользовательский экран, а вовсе даже на экран среды. Ясное дело, чтобы не создавать на экране кашу, лучше всего переключать экран по-глупому - при выполнении каждого оператора.

"Source debugging". Отладка в терминах исходной программы. Переключатель выглядит несколько чужеродным в меню "Debug", так как все остальные команды меню используются во время отладки, эта же - перед сборкой. Здесь достаточно сказать, что для использования встроенного отладчика переключатель "Source debugging" в момент вызова сборки программы должен находиться в положении "On". Более подробно назначение этого переключателя рассмотрено в главе 4.

Прежде, чем рассматривать последнее из меню команд отладчика, поговорим о том, как устроено окно "Watch". Окно это позволяет в процессе выполнения программы следить за значениями довольно широкого класса выражений (в том числе и отдельных переменных): ограничения - выражения не должны содержать вызовов функций и макросов, то есть имен, определенных с помощью конструкции #define. Потенциально окно может содержать любое количество выражений, каждое из них занимает отдельную строку. Переход в окно "Watch" из окна редактора и обратно выполняется клавишей F6, из окна "Messages" и обратно - Alt-F6. В каждый момент одно из выражений является текущим; при нахождении в окне оно выделено подсветкой, в противном случае (если окно находится на экране) слева от текущего выражения стоит пометка - символ, называемый "пуля" (bullet). Команды, доступные пользователю в окне "Watch":

- движение по строкам: стрелки вверх и вниз, PgUp и PgDn (перемещение на столько строк, сколько занимает окно - в однооконном или двухоконном режиме), Ctrl-E - на строку вверх, Ctrl-X - на строку вниз;

- движение внутри строки. Если выражение вместе со своим значением занимает больше, чем 78 символов (длина видимого участка строки окна), показывается только его часть, обычно начальная. Если такая длинная строка - текущая, видимый ее участок можно сдвигать влево и вправо (стрелки влево и вправо, а также Ctrl-S и Ctrl-D), можно передвигаться в начало строки (Home и Ctrl-A) или ее конец (End и Ctrl-F). Вообще длина строки в окне "Watch" ограничена, так что значение, например, достаточно длинной символьной строки вполне может в ней не поместиться. В этом случае строка окна содержит только само выражение и 255 начальных символов изображения его значения. Если при этом попытаться переместиться в конец строки, на экране будет показаны последние символы из этих 255-и, а не последние символы самого значения;

- добавление выражения после текущего (Ins). Открывается окно для ввода выражения. Правила ввода выражений, общие для окна "Watch" и команды "Evaluate", приведены в самом конце главы;

- удаление текущего выражения (Del или Ctrl-Y). Текущим становится следующее по порядку выражение;

- редактирование текущего выражения (Enter). Предлагается окно, как и при вводе выражения. Изначально в окне записано текущее выражение, которое будет заменено новым. Как обычно, для того, чтобы приступить к редактированию выражения, нужно сначала нажать одну из клавиш дополнительной клавиатуры.

"Break/watch". В этом меню собраны вместе команды, управляющие работой с окном "Watch" (отчасти дублирующие только что перечисленные,

доступные непосредственно в окне) и с точками прерываний. Команды работы с выражениями окна "Watch":

"Add watch" (Ctrl-F7). *Добавить выражение в окно "Watch"*. Эффект команды почти тот же, что при нажатии Ins в окне "Watch". Небольшое, но ценное, добавление: если курсор окна редактора в момент нажатия Ctrl-F7 находится на идентификаторе, этот идентификатор помещается в окно ввода выражения. Дополнительное удобство: нажимая стрелку вправо, можно добавлять в окно ввода символы, следующие за идентификатором в тексте программы. Например, чтобы ввести выражение из текста `arg[5]+j`, достаточно подвести курсор к любой букве слова `arg`, нажать Ctrl-F7, затем пять раз стрелку вправо и Enter.

"Delete watch". *Удалить текущее выражение окна "Watch"*. Эффект команды аналогичен нажатию Del в окне "Watch". Чтобы команда сработала, нужно, чтобы окно было видно на экране.

"Edit watch". *Редактировать текущее выражение окна "Watch"*. То же, что Enter внутри окна.

"Remove all watches". *Удалить все выражения, помещенные в окно "Watch"*.

Оставшиеся команды меню "Break/watch" посвящены точкам прерываний. Об этих точках уже много раз упоминалось без лишних пояснений в надежде, что каждый, кому приходилось отлаживать программы при помощи более-менее развитого отладчика, знает, что такое breakpoint. На всякий случай все-таки скажем здесь пару объясняющих слов. Точка прерывания в отладчике среды Турбо Си - это строка текста, при достижении которой (перед выполнением операторов, содержащихся в строке) выполнение программы приостанавливается, на экран выводится файл, содержащий эту строку, и среда ожидает от пользователя команд. Для того, чтобы строку можно было объявить точкой прерывания, необходимо выполнение следующих условий:

- строка должна содержать конструкции, для которых сгенерирован выполнимый код. Это может быть оператор, заголовок определения функции, описание локальных переменных с инициализацией, выражение. Точкой прерывания можно объявить даже строку, содержащую один-единственный плюс (минус, звездочку и так далее). Останавливаться выполнение будет перед началом вычисления выражения, этот плюс содержащего. Нельзя установить точку прерывания на строке, содержащей только описание переменной (функции) вне какой-либо функции, описание локальных без инициализации, на пустой строке и тому подобном. Помните, что оптимизация переходов, проводимая компилятором при работе в режиме "Options/Compiler/Optimization /Jump optimization...On", может исказить картину привязки кода к строкам исходного текста и отладчик может отказаться устанавливать точку прерывания в совершенно невинных на первый взгляд местах;

- файл, содержащий строку, должен входить в состав программы;
- программа должна быть собрана в режиме "Debug/Source debugging...On".

О неверно установленных точках прерываний среда сообщает либо сразу же при попытке это сделать, либо перед началом выполнения программы.

Одновременно может быть установлено не более 21 точки прерываний.

Команды работы с точками прерываний: "Toggle breakpoint" (Ctrl-F8). Установить/отменить точку прерывания. Объявляет точкой прерывания строку, на которой находится курсор окна редактора. Строка выделяется подсветкой и будет показываться выделенной до тех пор, пока данная точка не будет отменена. Отмена точки прерывания производится тем же самым нажатием Ctrl-F8.

"Clear all breakpoints". Удалить все точки прерываний.

"View next breakpoint". Вывести следующую точку прерывания. Команда позволяет просматривать все установленные в данный момент точки прерываний в том порядке, в каком они были заданы. Просмотр циклический: следом за точкой, установленной последней, команда показывает ту, которая была установлена первой.

Теперь перейдем к отложенному ранее разговору о команде "Debug / Evaluate" и заодно - об окне "Watch", использующем те же соглашения о формате вывода значений выражений.

Как уже было сказано, команда "Evaluate" (Ctrl-F4) используется для просмотра значений выражений и изменения значений переменных. Как это выглядит? На экран выводится окно, состоящее из трех частей: поле ввода выражения, поле вывода значения и поле задания нового значения. Если в момент нажатия Ctrl-F4 курсор в окне редактора располагался на идентификаторе, в поле ввода выражения изначально будет записан этот идентификатор. Точно так же, как в команде "Add watch" (см. выше), при нажатии стрелки вправо в поле ввода выражения будут помещаться соседние с идентификатором символы. Для изменения уже имеющегося в поле выражения можно сначала нажать одну из клавиш дополнительной клавиатуры. В выражении можно использовать любые переменные (глобальные и локальные), доступные в точке, где в данный момент находится управление. Можно использовать даже недоступные по правилам Си переменные, например, локальные для другого модуля или даже для другой функции из числа активных (то есть функций, входящих в текущую цепочку вызовов). Общий синтаксис обозначения переменных таков (квадратными скобками отмечены необязательные элементы):

[имя модуля .] [имя функции .] имя переменной .

Если опущено имя модуля, подразумевается текущий модуль, если опущено имя функции - переменная считается глобальной в указанном модуле или доступной в текущей функции. При вводе выражений не допускается использование операции "запятая", по причинам, объясняемым ниже.

После окончания ввода выражения нажимается Enter, и в поле вывода появляется его значение. Если по каким-либо причинам значение не может быть вычислено (выражение содержит обращение к функции, макросу или просто ошибочно с точки зрения языка), в поле вывода записывается диагностика. Выражение при этом остается в поле ввода, и может быть отредактировано обычным образом. Значение объекта можно изменить, если выражение представляет обозначение объекта, например, простую переменную, элемент массива и тому подобное - словом, то, что в англоязычной литературе по Си называется lvalue, а в нашей - "именующим выражением", "адресным выражением" или как-то похоже. Для этого стрелками или клавишей табуляции переводят курсор в нижнюю часть окна - поле нового значения - и вводят это самое новое значение. В качестве нового значения может выступать любое корректное с точки зрения языка выражение соот-

ветствующего типа, не содержащее обращений к функциям и макросов. На всякий случай отметим, что при задании нового значения не работает побочный эффект некоторых операций, таких, как ++ и ей подобные.

Другим способом изменения значения объекта может служить вычисление выражения-присваивания с этим объектом в левой части. Здесь уже побочный эффект работает. Может быть, имеет смысл предупредить: возможностью изменения значений нужно пользоваться с осторожностью, ибо человеко-веками программистской практики показано: неразумное вторжение в ход выполнения программы редко приводит к положительным результатам, а гораздо чаще - к полной путанице. По этой же причине, видимо, в отладочные средства Турбо Си не включена возможность "ручной" передачи управления в другую точку программы.

Как команда "Evaluate", так и окно "Watch", обладают еще одним полезным свойством: значения вычисляемых выражений можно выводить в различных форматах. В общем виде задание на вычисление выражения имеет вид (элементы, заключенные в квадратные скобки, могут отсутствовать):

выражение [, [повторитель] формат] .

Обратите внимание на запятую: теперь становится ясно, почему в выражении нельзя использовать операцию "запятая" (глубокого смысла ее использовать, конечно, и нет).

Если опущен формат, среда сама выбирает способ представления значения, исходя из его типа:

- значения целых типов выводятся как десятичные целые числа;
- значения типа `char *` выводятся как строки символов;
- значения других ссылочных типов выводятся в машинном представлении "сегмент:смещение", где сегмент и смещение представлены шестнадцатеричными константами (отметим, записываемыми без префикса `0x`, то есть не по правилам Си). Сегмент также может быть записан в виде символического имени (`DS`, `CS`, `SS`). Если указатель ссылается на переменную, имя которой известно отладчику, значение указателя может быть выведено как &идентификатор;
- значения типов `float` и `double` выводятся как плавающие константы;
- значения структур и объединений выводятся в виде перечня значений всех полей через запятые, заключенного в фигурные скобки.

Будучи задан, формат определяет способ интерпретации значения объекта, используемый при выводе, например, формат `h` задает вывод значения в шестнадцатеричном виде. Перечислим допустимые форматы:

`c` - символ кода ASCII. Целые значения от 0 до 255 выводятся как символы с соответствующими кодами; если значение целого 256 и больше, выводится целое. Для структур и объединений в формате "`c`" выводятся все поля через запятую. Плавающие типы и указатели всегда выводятся в обычном формате;

`d` - десятичное целое. Значения типа `char` выводятся как целые числа (значения остальных типов - как обычно);

`f [n]` - число с плавающей запятой. Влияет на представление значений типов `float` и `double`. Число `n` задает количество десятичных цифр в представлении мантиссы;

`h` или `x` - шестнадцатеричное число. Целые значения и указатели выводятся в шестнадцатеричном виде;

m - дамп памяти. Память, занимаемая значением, выводится в виде последовательности байтов, каждый из которых представлен шестнадцатеричным числом;

p - длинный указатель. Указатель выводится в формате "сегмент:сместение" (в дополнение к идентификатору переменной, на которую он, возможно, ссылается);

r - структура. Перечисляются все поля структуры с указанием их идентификаторов - в виде "поле:значение";

s - строка. Значения типа `char` и целые от 0 до 255 выдаются как символы кода ASCII, причем управляющие символы изображаются при помощи Escape-последовательностей Си: например, символ табуляции (код 9) будет выглядеть как `'\t'`, а символ с десятичным кодом 18 - как `'\x12'`.

Повторитель можно задавать только в случае, если выражение обозначает какой-либо объект, например, переменную. Используется он для того, чтобы задать количество выводимых значений. С его помощью можно узнать не только значение переменной, но и значения, хранящиеся в памяти рядом, в частности, возможен вывод любого отрезка массива или просто содержимого какого-то участка памяти. По поводу повторителей можно говорить еще много слов, но ничто не объяснит его работу так коротко и понятно, как пример. В иллюстрации нуждается также и текст о форматах выражений, так что закончим главу набором примеров выражений, применимых в команде "Evaluate" и окне "Watch".

Предположим, имеются описания:

```
int ch[4];
float r = 1.2345;
struct {int a;
       char b;
} str= {128, '\t'}, *pstr=&str;
```

и элементам массива `ch` присвоены значения от 0 до 255 по порядку (элемент с индексом 0 равен 0, с индексом 1 - 1, и так далее). Вот как будут вычислены значения разных выражений, использующих `ch` (в левой колонке - выражения, в правой - значения, которые будут выведены в окне "Watch" или в поле вывода "Evaluate"):

<code>ch[3]</code>	3
<code>(char*)ch+2</code>	<code>"\x1"</code>
<code>ch</code>	<code>{ 0, 1, 2, 3, }</code>
<code>ch[1],5h</code>	<code>0x1, 0x2, 0x3, 0x4, 0x5</code>
<code>*ch,8m</code>	<code>00 00 01 00 02 00 03 00</code>
<code>r</code>	<code>1.2345</code>
<code>r,f3</code>	<code>1.23</code>
<code>str</code>	<code>{ 128, '\t' }</code>
<code>str,cr</code>	<code>{ 'A', ' ' }</code>
	(в кавычках будет выведен символ ASCII с кодом 9)
<code>str,rs</code>	<code>{ a:'A', b:'\t' }</code>
<code>str.b,d</code>	9
<code>pstr</code>	<code>&str</code>
<code>pstr,p</code>	<code>DS:FFCC &str</code>
	(вместо <code>DS:FFCC</code> возможен какой-то другой адрес) .

7. РАБОТА ИЗ КОМАНДНОЙ СТРОКИ.

Те из читателей, кто изучал главу 2, возможно, помнят рассуждение о том, как хорошо работать в интегрированной среде Турбо Си в сравнении со средой ДОС. Не отказываясь от всего, что было сказано, позволим себе заметить, что иногда (к счастью, не слишком часто) приходится покидать гостеприимную среду и заниматься вызовом программ Турбо Си из командной строки. Делается это чаще всего из-за суровой необходимости (обычный случай - создаваемая программа не помещается в памяти вместе со средой), но есть и любители; в одном из руководств фирмы Borland International такие люди изящно названы *commandline aficionados* (возьмем грех на душу, переведа это как *"рыцари командной строки"*). Кроме того, утилита MAKE, обычно используемая при программировании больших проектов на Си в среде ДОС, обладает гораздо более богатыми возможностями, чем аналогичная по назначению команда среды, и при достаточно сложной структуре проекта выгоды от ее использования вполне могут перевесить удобства работы в среде.

В этой главе мы поговорим о том, как запускать автономный компилятор ТСС и автономный линкер TLINK, рассмотрим их опции (разговор об опциях, в основном, сведется к установлению соответствия между ними и переключателями меню среды), а также изучим утилиту MAKE.

Автономный компилятор Турбо Си ТСС по своим функциям отличается от команды среды "Compile": он способен транслировать за один вызов несколько файлов, воспринимает файлы с текстом на ассемблере (и вызывает для их трансляции ассемблер - возможность, в среде вообще отсутствующая) и может вызвать линкер для сборки выполнимой программы. Общий синтаксис вызова ТСС таков:

ТСС [опция опция ...] имяфайла имяфайла ...

Имена файлов, записываемые в команде после опций, могут задавать компилятору не только источники компилируемых текстов на Си, но и файлы других типов, в зависимости от расширения:

- файлы, расширение которых не указано, либо указано, но отличное от .ASM, .OBJ и .LIB, транслируются как файлы с текстами на Си (по умолчанию принимается расширение .C);

- файлы с расширением .OBJ включаются в выполнимую программу при сборке;

- файлы с расширением .LIB используются при сборке как библиотеки объектных программ;

- файлы с расширением .ASM транслируются как тексты на ассемблере: для этого вызывается ассемблер TASM (версия Турбо Си 1.5 вызывает MASM; можно также задать вызываемый ассемблер при помощи опции).

OBJ-файлы, полученные при трансляции текстов на Си и на ассемблере, также используются при сборке выполнимой программы.

Опции управляют работой компилятора и, если подразумевается сборка, линкера. Каждая опция начинается с минуса и должна быть отделена от остальных опций (или от первого имени файла, или от команды ТСС) по крайней мере одним пробелом. Для опций, представляющих собой переключатели, после имени опции ставится плюс (переключатель в положении "On") или минус ("Off"), например, -C+, -A-. Если после имени опции переключателя не стоит ни плюса, ни минуса, подразумевается плюс. Некоторые из опций должны содержать имя файла. При этом оно записывается вплотную к названию опции, например -efont задает имя выполняемого файла FONT.EXE. Так же записываются и числа, являющиеся параметрами некоторых опций. Строчные и прописные буквы в названиях опций различаются, так что -a и -A - это разные вещи.

Заданные опции отрабатываются слева направо; если встречается повторное использование опции в одной командной строке, действительным считается последнее из использований. Исключения составляют опции -D, -I и -L, эффект которых при многократном применении складывается (см. описание этих опций ниже в этой главе).

Большинство опций, управляющих работой компилятора и линкера, имеет аналоги в меню интегрированной среды. Чтобы не тратить даром бумагу и время читателей, приведем лишь таблицу соответствия для этих опций, а подробно поговорим лишь о тех, что уникальны для ТСС.

В таблице для краткости меню среды "Options/Compile", "Options/Linker" и "Options/Directories" обозначаются как "O/C", "O/L" и "O/D" соответственно. Выражение "по умолчанию" здесь надо понимать как "если опция не указана".

Позиция меню	Опция командной строки интегрированной среды
"Compile/Compile to OBJ"	-c (-c- компиляция и сборка, -c+ только компиляция; по умолчанию компиляция со сборкой)
"O/C/Model"	-m# (# - обозначение модели памяти: t- Tiny, s - Small, m - Medium, c - Compact, l - Large, h - Huge; по умолчанию - Large)
"O/C/Defines"	-D определения (можно перечислять определения через точку с запятой; нельзя использовать пробелы внутри определений; эффект нескольких употреблений опции

-D складывается)

"O/C/Code generation/
Calling convention"

-p
(-p- C,
-p+ Pascal;
по умолчанию - C)

"O/C/Code generation/
Instruction set"

-1
(-1- 8088/8086,
-1+ 80186/80286;
по умолчанию 8088/8086)

"O/C/Code generation/
Floating point"

-f
(-f- None,
-f+ Emulation,
-f87 8087/80287;
по умолчанию Emulation)

"O/C/Code generation/
Default char type"

-K
(-K- Signed,
-K+ Unsigned;
по умолчанию Signed)

"O/C/Code generation/
Alignment"

-a
(-a- Byte,
-a+ Word;
по умолчанию Byte)

"O/C/Code generation/
Generate underbars"

-u
(-u- Off,
-u+ On;
по умолчанию On)

"O/C/Code generation/
Merge duplicate strings"

-d
(-d- Off,
-d+ On;
по умолчанию Off)

"O/C/Code generation/
Standard stack frame"

-k
(-k- Off,
-k+ On;
по умолчанию On)

"O/C/Code generation/
Test stack overflow"

-N
(-N- Off,
-N+ On;
по умолчанию On)

"O/C/Code generation/
Line numbers"

-y
(-y- Off,
-y+ On;
по умолчанию Off)

"O/C/Code generation/ OBJ debug information"	-v (-v- Off, -v+ On; по умолчанию Off)
"O/C/Optimization/ Optimize for"	-G (-G- Size, -G+ Speed; по умолчанию Size)
"O/C/Optimization/ Use register variables"	-r (-r- Off, -r+ On, по умолчанию On)
"O/C/Optimization/ Register optimization"	-Z (-Z- Off, -Z+ On; по умолчанию Off)
"O/C/Optimization/ Jump optimization"	-O (-O- Off, -O+ On, по умолчанию Off)
"O/C/Source/ Identifier length"	-i# (# - число значащих символов; по умолчанию 32)
"O/C/Source/Nested comments"	-C (-C- Off, -C+ On; по умолчанию Off)
"O/C/Source/ ANSI keywords only"	-A (-A- Off, -A+ On; по умолчанию Off)
"O/C/Errors/ Errors : stop after"	-j# (# - максимальное число ошибок; по умолчанию 25)
"O/C/Errors/ Warnings : stop after"	-g# (# - максимальное число предупреждений; по умолчанию 100)

"O/C/Errors/ Display warnings"	-w (-w- Off, -w+ On; по умолчанию On)
"O/L/Map file"	-M (-M- Off, -M+ Detailed; по умолчанию Off)
"O/D/Include directories"	-Идиректории (можно перечислять директории через точку с запятой; эффект нескольких применений опции -I складывается)
"O/D/Library directories"	-Лдиректории (можно перечислять директории через точку с запятой; эффект нескольких применений опции -L складывается)
"O/D/Output directory"	-ндиректорий (по умолчанию - текущий директорий)
"Debug/Source debugging"	-v (-v- None, -v+ On; по умолчанию None)

Заметим, что опция -v выполняет двойную функцию: при задании -v+ компилятор генерирует отладочную информацию в OBJ-файлах (как при переключателе "Options/Compiler/Code generation/OBJ debug information...On"), и при сборке она помещается в EXE-файл ("Debug/Source debugging...On").

Отдельные предупреждающие сообщения также могут быть включены или подавлены с помощью опции командной строки. Для этого используется опция -w с условным обозначением предупреждения, например, опция -w+rvl (тот же эффект имеет -wrvl) задает включение обычно не выдаваемого предупреждения "Function should return a value". Приведем список обозначений предупреждений, использующихся опцией -w (звездочками отмечены предупреждения, выдаваемые по умолчанию):

"O/C/Errors/Portability warnings"

"Non-portable pointer conversion"	* rpt
"Non-portable pointer assignment"	* apt
"Non-portable pointer comparison"	* cpt
"Constant out of range in comparison"	* rng
"Constant is long"	cln
"Conversion may lose significant digits"	sig
"Mixing pointers to signed and unsigned char"	ucp
"O/C/Errors/ANSI violations"	
"'ident' not part of structure"	* str
"Zero length structure"	* zst
"Void functions may not return a value"	* voi
"Both return and return of a value used"	* ret
"Suspicious pointer conversion"	* sus
"Undefined structure 'ident'"	* stu
"Redefinition of 'ident' is not identical"	* dup
"Hexadecimal or octal constant too large"	* big
"O/C/Errors/Common errors"	
"Function should return a value"	rvi
"Unreachable code"	* rch
"Code has no effect"	* eff
"Possible use of 'ident' before definition"	* def
"'ident' is assigned a value which is never used"	* aus
"Parameter 'ident' is never used"	* par
"Possibly incorrect assignment"	* pia
"O/C/Errors/Less common errors"	
"Superfluous & with function or array"	amp
"'ident' declared but never used"	use
"Ambiguous operators need parentheses"	amb
"Structure passed by value"	stv
"No declaration for function 'ident'"	nod
"Call to function with no prototype"	pro

Теперь перейдем к рассмотрению опций ТСС, аналогов которым в среде Турбо Си не имеется.

-В. Опция сообщает компилятору, что в транслируемом файле (файлах) имеются ассемблерные вставки, для трансляции которых нужно будет вызвать TASM (или другой ассемблер, заданный при помощи опции -Е).

-Еимя. Задает имя исполняемого файла, который вызывается для трансляции ассемблерных вставок в текст на Си. Имя может быть задано вместе с путем и/или расширением.

-енмя. Задает имя файла, в который будет помещена выполняемая программа. Если не указано расширение, им будет .EXE. При отсутствии опции -е EXE-файл получает название по имени первого из файлов (С или OBJ), заданных в качестве параметров ТСС.

-оимя. Задаёт имя OBJ-файла, получаемого в результате компиляции. Используется при компиляции отдельных файлов.

-S. При задании этой опции компилятор генерирует не OBJ-файл, а файл с текстом на ассемблере (и расширением .ASM). Очень интересная возможность для любителей "вылизывать" программы до максимальной эффективности.

-Uимя. Опция, парная к опции -D (определение макроимен) и обратная к ней. После того, как при помощи опции -D в командной строке были определены какие-то имена, определение любого из них можно отменить, указав это имя в опции -U. Это похоже на использование в исходном тексте директивы препроцессора #undef. Отметим, что в среде Турбо Си такая возможность отсутствует, да и не очень-то нужна. В командной же строке она может быть необходима, если используется файл конфигурации (см. ниже), в котором определены какие-то имена.

-I. Эта опция позволяет передать линкеру TLINK, вызываемому компилятором TCC для сборки выполнимой программы, любые из его опций. Для подавления используется форма -I-x, где x - одна или несколько опций линкера. Для включения опций можно писать -I+x или просто -Ix, здесь x также обозначает любой набор опций линкера.

Разбор опций и параметров TCC завершим примером. Пусть у нас есть программа, состоящая из трех модулей на Си - MAIN, COMPUTE и IO. Кроме того, в EXE-файл нужно включить уже имеющийся файл SCREEN.OBJ и ассемблерный модуль BIOSINT.ASM, который нужно предварительно ассемблировать. При сборке используется библиотека LONGMATH.LIB, расположенная в текущем директории, и стандартные библиотеки Турбо Си, находящиеся в директории C:\TC\LIB. Стандартные хедер-файлы расположены в директории C:\TC\H. Полученный EXE-файл нужно назвать CALC.EXE. Вот как будет выглядеть вызов автономного компилятора:

**TCC -LC:\TC\LIB -IC:\TC\H -eCALC MAIN COMPUTE IO
SCREEN.OBJ BIOSINT.ASM LONGMATH.LIB .**

Прикиньте, насколько весело набирать этот текст каждый раз, когда нужно проделать подобную работу. Даже если бы файлов было бы меньше библиотеки и стандартные хедеры нужны всегда. Так что же, каждый раз набирать опции -L и -I? Конечно, нет. Как раз на этот случай предусмотрена возможность задавать опции автономного компилятора не в командной строке, а в файле, носящем имя TURBOC.CFG и называемом файлом конфигурации компилятора. Файл конфигурации компилятора - обычный текстовый файл. Опции в этом файле записываются точно в том же виде, как если бы они задавались в командной строке, причем не обязательно на одной строке.

Когда запускается автономный компилятор, он ищет файл TURBOC.CFG в текущем директории, и если не находит - то в директории, из которого был вызван TCC (последнее возможно только в версиях DOS с номерами 3.0 и выше). Опции, заданные в файле конфигурации, присоединяются к опциям, заданным в командной строке (если такие есть), причем слева, так что опции командной строки "накрывают" опции, взятые из файла, тем самым пользователю дается возможность изменить любые "стандартные" установки. Исключение - опции -L и -I, содержащие перечень директорий для поиска соответственно библиотек и стандартных хедер-фай-

лов. Эти опции, будучи считанными из файла конфигурации компилятора, присоединяются к опциям командной строки не слева, как остальные, а справа. В итоге поиск файлов (библиотек или хэдеров) происходит сначала в директориях, заданных в командной строке, и лишь затем - в директориях, заданных в опциях из файла конфигурации.

Добавим, что есть очень удобный способ составлять файл конфигурации компилятора автоматически, причем так, чтобы установки, задаваемые этим файлом, копировали установки, записанные в файле конфигурации среды Турбо Си. Это достигается при помощи утилиты TCCONFIG, использование которой подробно описано в следующей главе.

Так же, как и в среде Турбо Си, линкер чаще всего вызывается автоматически по успешном завершении компиляции, но есть возможность вызвать его как автономную программу. Называется эта программа TLINK и представляет из себя стандартный линкер, который за счет исключения некоторых удобств и экономии на диагностике сделан очень компактным и быстрым.

Синтаксис вызова TLINK:

TLINK objfiles , exefile , mapfile , libfiles

Здесь objfiles обозначает список имен OBJ-файлов, участвующих в сборке выполняемой программы, exefile - имя создаваемого EXE-файла, mapfile - имя файла, куда будет записана карта сборки, libfiles - список библиотек объектных программ, используемых при сборке. Имена файлов в списках разделяются пробелами. На месте любого пробела в списке параметров может быть указана опция. Опция начинается с символа '/' (слэш), за которым следует однобуквенное имя опции.

Имена EXE-файла и файла карты, а также и библиотеки, если они не нужны, можно опускать (запятые же - нельзя). Если опущено имя EXE-файла, создаваемый файл будет называться по имени первого из списка OBJ-файлов (расширение .EXE или, если задана опция /t, .COM). Если опущено имя файла карты сборки и генерация файла не подавлена опцией /x, файл карты будет иметь имя EXE-файла и расширение .MAP.

Как и компилятор, линкер имеет способ облегчить ввод длинных командных строк - при помощи так называемых "файлов подстановки" (response files). В любом месте командной строки можно использовать конструкцию @имяфайла - это будет означать, что продолжение командной строки линкер должен считать из указанного текстового файла (в имени файла можно указывать путь). Когда файл подстановки заканчивается, линкер продолжает разбирать командную строку со следующего за именем файла символа. В одной строке можно использовать любое число подстановок. Наиболее очевидное использование этой возможности - оформление в виде файлов подстановок списков OBJ-файлов и списков библиотек. Вызов TLINK в этом случае может принять вид

TLINK @OBJs,,@LIBS

если файл OBJs содержит список OBJ-файлов, участвующих в сборке, а файл LIBS - список используемых библиотек.

Содержимое файла подстановки может быть записано как в одну строку, так и в несколько. Для того, чтобы продолжить запись параметров на следующую строку, в конце строки ставят плюс. Другой способ записи

- начинать каждую из четырех компонент задания с новой строки (плюс в этом случае не используется). При этом роль запятой выполняет символ конца строки, так что некоторые из запятых нужно опустить. Вот как, например, может выглядеть файл подстановки, содержащий целиком задание на сборку:

```
MAIN.OBJ +  
COMPUTE+  
IO SCREEN BIOSINT  
CALC,  
LONGMATH.LIB
```

Поясним: запятая, оставленная после имени EXE-файла CALC, говорит о том, что имя файла с картой сборки выбирается по умолчанию. Толкование остальных нюансов предоставим читателю в качестве упражнения.

Если приведенный файл подстановки называется, например, LINKCMD, то линкер может быть вызван просто командой

```
TLINK @LINKCMD
```

Все это относится к способу вызова TLINK вообще, для выполнения стандартной сборки программ, к Турбо Си отношение иметь не обязанных. Для того, чтобы правильно собирать при помощи TLINK программы, написанные на Турбо Си, нужно соблюдать кое-какие правила (их, конечно, соблюдает и TCC, вызывая TLINK). Вот эти правила:

1. Первым в списке OBJ-файлов должен стоять файл C0x, где x - символ, обозначающий модель памяти, в которой скомпилирована программа (t - tiny, s - small, m - medium, c - compact, l - large, h - huge). Файл C0x содержит код инициализации среды выполнения программ Турбо Си, на него передает управление ДОС, и он же осуществляет возврат в ДОС по окончании работы программы (как видим, приведенный в качестве примера файл подстановки был, строго говоря, ошибочным).

2. В зависимости от режима трансляции, связанного с обработкой вещественных данных (переключатель среды "Options /Compiler/Code generation/ Floating point" или опция командной строки TCC -f), должны быть включены стандартные библиотеки EMU (переключатель в "Emulation" или опция -f+) либо FP87 (переключатель в "8087/80287" или опция -f87). Кроме того, в этих двух случаях необходимо включение стандартной библиотеки MATHx (x - символ модели памяти; модели tiny и small используют одну и ту же библиотеку MATHS). Если в программе нет работы с плавающей точкой, включение перечисленных библиотек не обязательно.

3. Если в программе используется хотя бы одна графическая функция Турбо Си, список библиотек должен содержать стандартную библиотеку GRAPHICS.

4. После библиотек, включенных во исполнение условий 2 и 3, в списке библиотек должна стоять стандартная библиотека Sx (x - символ модели памяти), содержащая стандартные функции и переменные Турбо Си.

5. Должно быть задано имя EXE-файла (иначе файл будет называться C0x.EXE по имени первого из OBJ-файлов - см. условие 1).

Само собой разумеется, стандартные библиотеки и файл C0x.OBJ должны задаваться с указанием пути - иначе линкер их не найдет.

Перейдем теперь к рассмотрению опций TLINK. Как и у ТСС, большая часть опций дублирует позиции меню среды, на этот раз - "Options/Linker". Приведем и здесь таблицу соответствия этих опций позициям меню, а разговор поведем лишь об оставшихся ("Options/Linker" сокращаем как "O/L").

Позиция меню	Опция командной строки интегрированной среды
"o/L/Map file...Off"	/x
"O/L/Map file...Segments"	по умолчанию
"O/L/Map file...Publics"	/m
"O/L/Map file...Detailed"	/s
"O/L/Initialize segments...On"	/i
"O/L/Default libraries...Off"	/n
"O/L/Warn duplicate symbols...On"	/d
"O/L/Case-sensitive link...On"	/c
"Debug/Source debugging...On"	/v

При вызове TLINK из автономного компилятора ТСС всегда задается опция /с. Кроме того, если компилятору задана опция -v+ (полная отладочная информация), она (в виде /v) передается и линкеру.

Теперь - опции, аналогов которым в меню среды нет.

/1. Включение в файл карты сборки информации о номерах строк в исходном тексте. Информация включается только для модулей, которые транслировались в среде в режиме "Options/ Compiler /Code generation/Line numbers...On" или автономным компилятором с опцией -y+.

/3. 32-битовый код. Используется, если один или более из модулей, участвующих в сборке, содержат 32-битовый код для процессора 80386 (и, очевидно, написан на соответствующем ассемблере). Вряд ли в ближайшее время этой опции грозит большая популярность среди советских программистов.

/v. Включение в EXE-файл отладочной информации. Действует только для модулей, транслированных в среде в режиме "Options/ Compiler/Code generation/OBJ debug information... On" или автономным компилятором с опцией -v+.

/e. Отключение использования расширенного словаря. Расширенный словарь (Extended Dictionary) - вспомогательная структура библиотеки объектов программ, позволяющая ускорить работу линкера Турбо Си. Все стандартные библиотеки содержат расширенный словарь, можно его добавить и в другие библиотеки. Причины, по которым иногда требуется отключить использование расширенного словаря, могут быть связаны с недостатком памяти (плата за скорость сборки с использованием словаря), либо с тем фактом, что при использовании словаря линкер игнорирует любую отладочную информацию, хранящуюся в библиотеке.

/t. Создание COM-файла. При использовании этой опции линкер создаст вместо EXE-файла файл с расширением COM - тоже выполнимый, но оформленный по немного другим правилам. COM- файл может быть создан только для программ, которые транслировались в модели памяти tiny.

При программировании больших проектов, и не только на Си, большую помощь может оказать утилита MAKE, в свое время счастливо изо-

бренная неким М.Фелдманом. Смысл этой утилиты в следующем: пользователь описывает свой проект путем перечисления составляющих его файлов, при этом каждому файлу сопоставляется набор условий, при выполнении которых его нужно считать устаревшим, и набор команд (например, команда ДОС). При работе MAKE проверяет эти условия и для устаревших файлов выполняет связанные с ними команды. Таким образом, хотя изначально утилита была придумана для поддержания согласованности проектов на Си, ни к Си, ни к какому-либо другому языку программирования она непосредственного отношения не имеет. С ее помощью вполне можно поддерживать не программный проект, а, скажем, каталог грампластинок или базу данных, хранящую информацию о московских пивных.

Утилита MAKE, поставляемая вместе с Турбо Си, весьма похожа на ту, что используется в системе UNIX. Описание проекта помещается в текстовый файл (будем называть его make-файлом) и состоит, в основном, из так называемых зависимостей - предложений вида

цели : [условия]

[команда]

[команда]

...

И цели, и условия - это списки имен файлов, разделенных пробелами. Команды - любые команды ДОС. При составлении make-файла важно помнить, что список целей обязан начинаться с первой позиции строки, в то время как команда всегда записывается с отступом хотя бы в один пробел. Если список целей и условий (также и команда) не помещается на одной строке, как признак продолжения строки используется символ '\ '.

Обработка make-файла командой MAKE состоит в следующем. Поочередно просматриваются все зависимости. Для каждой зависимости сравниваются даты последнего изменения целей и условий. Если дата/время последнего изменения хотя бы одного из файлов-условий больше, чем у любой из целей, или какой-либо файл-цель вообще отсутствует, MAKE поочередно вызывает все команды, заданные в зависимости. Подразумевается, что пользователь задал команды таким образом, чтобы они обновляли цели с учетом изменений, внесенных в условия. Примером такого обновления может служить перекомпиляция, если цель - OBJ-файл, а условия - файл с исходным текстом и все используемые им хедер-файлы.

Понятно, что если описать проект make-файлом, в котором задать по одной зависимости на каждый OBJ-файл, в качестве условий этих зависимостей задать соответствующие файлы с исходными текстами, а в качестве команд - вызов автономного компилятора, то любой запуск MAKE приведет проект в согласованное состояние (конечно, если перекомпиляции не обнаружат ошибок). Если еще добавить зависимость для EXE-файла, условиями которой будут все OBJ-файлы проекта и используемые библиотеки, а командой - вызов линкера, то такой make-файл будет соответствовать файлу проекта, составляемому для команды "Make" интегрированной среды. Однако такое использование MAKE, хотя и вполне достойно, но не позволяет судить о всей мощи этой утилиты. Возможностей у нее, конечно, гораздо больше, чем было до сих пор показано. Здесь мы не будем тратить время на подробное описание всех этих возможностей - это очень хорошо сделано в "Руководстве по Турбо Си", да и любое описание команды make системы UNIX сойдется - как уже говорилось, эти две команды - близкие родственники. Вместо этого просто перечислим возможности, сказав о каждой лишь

пару слов. Задача этого перечисления - привлечь внимание читателя к мощному, хотя и малоиспользуемому (мало кто хочет высовывать нос из среды?) аппарату поддержания согласованности программного проекта.

Выше уже были рассмотрены зависимости, составляющие основное содержание make-файла. Эти зависимости иногда называют явными - можно догадаться, что есть и неявные. Действительно, конструкция

расширение_условия.расширение_цели

[команда]

[команда]

...

называется неявной зависимостью и задает сразу класс зависимостей. Для любого из файлов, упомянутых в make-файле в качестве цели и имеющих расширение "расширение_цели" из какой-либо неявной зависимости, автоматически одним из условий становится файл с тем же именем и с расширением "расширение_условия" из той же неявной зависимости. При изменении файла-условия выполняются команды, приведенные в неявной зависимости.

Make-файл может содержать комментарии. Они начинаются с символа '#' и продолжаются до конца строки.

Команды, задаваемые в зависимостях, могут быть снабжены префиксами. Этих префиксов три (вернее, два с половиной): @ - говорит о том, что команду перед выполнением на экран выводить не нужно (обычно это делается), -N - задает прекращение работы MAKE в случае, если код возврата команды (это то же самое число, которое можно получить в hatch-файлах переменной errorlevel) больше, чем N, наконец, просто - (минус) предписывает не прекращать работу MAKE вне зависимости от кода возврата команды.

Для удобства записи и для повышения гибкости задания make-зависимостей используется аппарат макропеременных, похожий на такой же аппарат Си. Макропеременная вводится конструкцией имя_макро = текст. Имя макропеременной может содержать только латинские буквы и цифры; начинается всегда с буквы. Текст макроподстановки может содержать любые символы и заканчивается символом конца строки.

Макропеременные, определенные в make-файле, могут быть впоследствии использованы в тексте. Для подстановки значения макропеременной пользуются обозначением

\$ (имя_макро)

Помимо определяемых пользователем макропеременных имеется набор предопределенных макросов. Их перечень

\$d (имя_макро)

принимает значение 1, если макропеременная с указанным именем определена, и 0 - в противном случае. Используется в основном в директивах условной обработки (см. ниже);

\$* задает имя файла-цели с путем, но без расширения. Используется в командах, особенно полезно в неявных зависимостях;

\$< - то же, что \$*, но имя файла задается с расширением;

\$: - задает только путь файла-цели;

\$. - задает имя файла-цели с расширением (без пути);

\$& - только имя файла, без пути и без расширения.

MAKE использует директивы, похожие на директивы препроцессора Си: **!include**, **!if**, **!else**, **!elif**, **!endif**, **!error**, **! undef**. Как видим, отличается написание директив от аналогичных директив Си лишь первым символом - '!' вместо '#'. В остальном же действие этих директив и правила построения выражений в директивах **!if** и **!elif** в точности совпадают с Си.

В заключение приведем небольшой пример **make**-файла, в котором постараемся продемонстрировать использование перечисленных возможностей. Проект, который мы опишем, уже фигурировал выше в этой главе как основа для примера вызова автономного компилятора.

#####

CALC.MAK: make-файл для проекта CALC

неявная зависимость: каждый из **OBJ**-файлов зависит

от файла с тем же именем и расширением **.C**

отметьте использование макропеременной **opts** (опции

компиляции) и полного имени файла-цели

(предопределенная макропеременная **\$<**)

.C.OBJ;

tcc \$(opts) \$<

еще неявная зависимость: **OBJ**-файл зависит от

хэдер-файла с тем же именем и создается

заново при его изменении.

.H.OBJ:

tcc \$(opts) \$<

определение макропеременной, содержащей опции

для вызова **TCC**. При изменениях набора

используемых опций достаточно внести их только

в следующую строку **opts= -ml -f87 -c -v+**

Явные зависимости для всех файлов проекта **#** Заметьте, что для
всех **OBJ**-файлов не упоминаются ни имена

исходных **C**-файлов, ни хэдер-файлы с теми же именами :

это уже сделано при помощи неявной зависимости.

MAIN.OBJ : COMPUTE.H IO.H

tcc \$(opts) \$<

для файла **COMPUTE.OBJ** явных зависимостей не требуется

COMPUTE.OBJ :

IO.OBJ : BIOSINT.H SCREEN.H

tcc \$(opts) \$<

файл **SCREEN.OBJ** получается не трансляцией, а переписью с

архивной дискеты (только в том случае, если его нет в

текущем директории)

SCREEN.OBJ :

echo Вставьте дискету с файлом **SCREEN.OBJ** и нажми кнопку!

pause

>NUL copy a:screen.obj

файл **BIOSINT.OBJ** получается вызовом ассемблера

BIOSINT.OBJ : BIOSINT.ASM

следующая команда не выводится на экран (префикс **@**)

@ c:\tasm\tasm biosint

```
# если все прошло успешно и обновилась какие-то OBJ-файлы
# или библиотека, вызывается линкер
CALC.EXE : MAIN.OBJ COMPUTE.OBJ BIOSINT.OBJ IO.OBJ \
          SCREEN.OBJ          tlink
c0$(model) main compute biosint io screen , \
          calc, \ calc, \ longmath.lib @iclib
# как только EXE-файл создан, программа запускается
echo Проверь, не напорол ли чего!
calc.exe
#      конец
#####
```

Для вызова утилиты MAKE для обработки этого файла нужно набрать

```
make -fcalc.mak calc.exe
```

Задание в командной строке calc.exe говорит о том, что окончательной целью является построение именно этого файла (если этого не указать, целью будет считаться первая из указанных в файле целей, а именно main.obj). Опция -f задает имя make-файла (по умолчанию MAKE обрабатывает только файлы makefile и makefile. mak).

8. УТИЛИТЫ

Если взглянуть на содержимое дистрибутивных дискет Турбо Си (впрочем, многие ли из нас могут похвастаться, что когда-нибудь их видели...) или в директорий Турбо Си, если, конечно, при установке системы туда были переписаны все файлы, обнаружится, что кроме программных файлов, назначение которых хорошо известно любому - ТС. EXE (среда Турбо Си), ТСС.EXE (компилятор для вызова из командной строки), TLINK.EXE (автономный линкер) и MAKE.EXE (утилита для поддержания согласованности проекта), описанных в предыдущих главах, там есть еще какие-то программы, назначение которых не всегда очевидно, особенно для тех, кто не дает себе труда или не имеет возможности изучить "Руководство по Турбо Си", а то и просто запускать эти программы и посмотреть, что же они делают.

В этой главе мы разберемся с тем, какие из дополнительных программ поставятся для нашего использования и как с ними работать.

Всего мы рассмотрим 10 программ. Вот их список:

BGIOBJ - преобразование драйверов BGI в OBJ-файлы;

CINSTXFR - перенос установок среды из версии 1.5 в версию 2.0;

CPP - препроцессор Турбо Си;

GREP - поиск текста в файлах;

OBJXREF - построение таблицы перекрестных ссылок проекта;

TCCONFIG - преобразование файлов конфигурации;

TCINST - настройка среды Турбо Си;

THELP - резидентная система помощи;

TLIB - утилита обслуживания библиотек объектных файлов;

TOUCH - установка даты последнего изменения файлов.

Поскольку это совершенно все равно, будем рассматривать утилиты в алфавитном порядке.

8.1. BGIOBJ

Тем, кто работает с графическими функциями Турбо Си, хорошо знакома аббревиатура BGI - Borland Graphic Interface. Файлы с расширением .BGI, входящие в состав системы Турбо Си, содержат драйверы экрана, позволяющие работать с ним так, как это принято у фирмы Borland International. Файлы с расширением .CHR содержат экранные шрифты, которые можно использовать при работе с драйверами BGI. Обычная схема использования драйверов и шрифтов - динамическая загрузка их во время выполнения при помощи функции `initgraph`. Эта схема имеет недостаток: выполняемая программа оказывается "привязанной" к драйверам и шрифтам, которые она использует. При перемещении программы на другую ЭВМ придется копировать вместе с EXE-файлом программы еще и эти файлы, либо убедиться, что они имеются на новом месте. Кроме того, каждый раз

на загрузку драйверов и шрифтов тратится время выполнения программы. Чтобы голова обо всем этом не болела, можно включить необходимые файлы **BGI** и **CHR** в выполняемую программу, предварительно преобразовав их в формат **OBJ**-файлов утилитой **BGIOBJ**. Обращение к **BGIOBJ** выглядит просто:

BGIOBJ имяфайла

В качестве параметра указывается имя файла (**.BGI** или **.CHR**), который нужно преобразовать в **OBJ**-формат. Новый файл будет иметь то же имя и расширение **.OBJ**. Этот файл нужно включить в проект или в одну библиотек, используемых линкером. В программе описывается переменная с именем **xxx_driver** (для шрифтов - **xxx_font**; **xxx** обозначает имя преобразованного файла, например, для использования драйвера **EGAVGA** переменная будет называться **EGAVGA_driver**, для шрифта **gothic** - **gothic_font**). Переменная описывается как ссылка на функцию без параметров и возвращаемого значения, например:

```
extern void (*EGAVGA_driver) (void);
```

```
extern void (*gothic_font) (void);
```

Перед инициализацией графической системы (вызовом функции **initgraph**) драйвер нужно зарегистрировать при помощи функции **registerbgidriver** (шрифт - **registerbgifont**), с упомянутой переменной в качестве параметра.

Программа **BGIOBJ** имеет еще кое-какие возможности, связанные с тонкостями работы линкера с **OBJ**-файлами. Здесь обсуждать их не будем.

8.2. CINSTXFR.

Начальная конфигурация среды (установки меню, принимаемые, если при запуске не был найден файл конфигурации, цвета различных элементов экрана, размеры окон и так далее) хранится непосредственно в файле, содержащем программу среды (обычное название - **TC.EXE**). Эти установки могут быть изменены при помощи утилиты **TCINST**, которая описана ниже в этой главе. Чтобы при переходе к новой версии пользовательский набор установок не нужно было создавать заново, существует программа **CINSTXFR**. Она извлекает из среды Турбо Си версии 1.5 все эти установки и записывает их в среду версии 2.0. **CINSTXFR** требует два параметра: имя файла среды версии 1.5 и имя файла среды версии 2.0, любое может содержать маршрут. Пример вызова **CINSTXFR**:

```
CINSTXFR C:\TURBOC\TC.EXE TC2\TC.EXE
```

8.3. CPP.

Эта программа представляет собой автономный препроцессор для Турбо Си. Как правило, использование такого препроцессора нужно в случаях, когда компилятор сообщает об ошибке внутри сложного (обычно многоуровневого) макровывода. Если источник ошибки не ясен сразу и его обнаружению не помогает прямолинейное разворачивание макрокоманд "вручную", имеет смысл взглянуть на код, получаемый после обработки программы препроцессором. В отличие от других компиляторов, в которых работа

препроцессора - отдельный проход по тексту и обычно имеется опция, позволяющая выдать текст, полученный в результате этого прохода, в Турбо Си проходы препроцессора и собственно компиляция совмещены. Чтобы все же иметь возможность контролировать результаты применения препроцессора, и предлагается использовать утилиту CPP. Вызов ее выглядит так:

CPP [опции] имяфайла имяфайла ...

В качестве параметров CPP получает имена файлов с исходными текстами на Си. В именах файлов можно использовать метасимволы '*' и '?' - будут обработаны все файлы, имена которых подходят под заданную маску. Все файлы, указанные в командной строке вызова CPP, проходят обработку препроцессором: включаются все хэдер-файлы, отрабатываются и исключаются из текста все директивы препроцессора, вызовы макроопределений разворачиваются. Результат помещается в файл с именем исходного файла и расширением .I. В начале каждой строки файла-результата записывается имя файла, из которого взята строка и номер строки в этом файле. По этой причине файл-результат не может быть откомпилирован. Если все-таки хочется его откомпилировать, при вызове CPP можно использовать опцию -P, подавляющую вывод информации об источниках строк. Остальные опции CPP представляют собой подмножество опций автономного компилятора TCC - исключены лишь опции, относящиеся к управлению генерацией объектного кода и вызовом линкера: здесь они и ни к чему.

8.4. GREP.

Утилита, как и MAKE, заимствованная из системы UNIX. Могучий инструмент, позволяющий отыскивать заданную строку в текстовых файлах. При работе в Турбо Си обычно пригождается, когда нужно отыскать место определения или места использования какого-либо идентификатора (при работе с большими программами это бывает нужно довольно часто). Простейший способ использования GREP - команда вроде следующей:

GREP lostname *.C

Нужно отметить, что даже такая примитивная возможность сильно облегчает жизнь, в том числе в ситуациях, не имеющих к Турбо Си отношения. Однако "найди идентификатор во всех файлах" - лишь небольшая часть способностей GREP, а среди остальных ее умений немало достойных использования. Во-первых, GREP умеет искать строки, заданные в гораздо более общем виде: так называемые регулярные выражения. Во-вторых, эта утилита обладает набором опций, которые позволяют производить поиск на любой вкус. Синтаксис обращения к GREP:

GREP [опции] строка [имяфайла имяфайла ...]

Рассмотрим сначала, что такое регулярное выражение, затем - набор опций GREP.

Регулярное выражение - шаблон, задающий общий вид строк, которые должны быть найдены. Параметр GREP, задающий строку поиска, может быть либо просто строкой, каждый символ которой воспринимается буквально, либо регулярным выражением, в котором некоторые символы интерпретируются как "операторы". Для того, чтобы параметр GREP воспринимался как регулярное выражение, задается опция -r (по умолчанию параметр трактуется как простая строка). В случае, если простая строка содержит пробелы или символы табуляции (обычно трактуемые как ограничители

ли), ее можно заключить в двойные кавычки, и пробелы будут восприниматься как часть строки.

Регулярное выражение есть последовательность символов, заключенная в двойные кавычки, либо отдельный символ. Последовательность регулярных выражений также является регулярным выражением. Следующие из символов при обработке регулярного выражения интерпретируются особым образом:

^ - соответствует началу строки текста. Например, выражению **^qq** сопоставятся только те строки, которые содержат символы **qq** в первых двух позициях;

\$ - соответствует концу строки текста. Обратно символу **^**, так что **qq\$** сопоставятся только строки, в последних двух символах содержащие **qq**, а **^qq\$** - те из них, что состоят только из этих двух символов;

. - используется для обозначения любого символа. Так, **^.\$** найдет все строки, содержащие только один символ, неважно какой;

***** - повторитель. Задает повторение нуля или более раз выражения, вслед за которым записан. Так, с выражением **define** **** \$** сопоставятся строки, содержащие слово **define**, за которым следует (возможно, через какое-то число пробелов) конец строки. Напомним, пробел заключается в кавычки, чтобы он не был воспринят как конец выражения;

+ - тоже повторитель, но в отличие от ***** соответствует только ненулевому числу повторений выражения, после которого записан. Выражение **define** **+\$** будет соответствовать только тем строкам, в которых между словом **define** и концом строки есть хотя бы один пробел;

**** - как и в Си, используется для введения в выражение управляющего символа без управляющей нагрузки. Пример: **\.** соответствует именно точке, а не любому символу, как было бы без обратного слэша, а **** представляет сам обратный слэш; при помощи этого символа можно также вводить в выражение пробелы (например, следующей записью из трех символов **\ +** задается любая непустая последовательность пробелов);

[] - в квадратные скобки заключают последовательность символов, интерпретируется которая как "один из указанных". Другая форма - последовательность начинается с символа **^** - означает "любой, кроме указанных". Примеры: **[0123456789]** обозначает любую из десятичных цифр, а **[^{}]** - любой символ, кроме фигурных скобок. Для сокращения записи можно использовать "диапазоны", записываемые как первый символ диапазона, вслед за ним минус, вслед за ним - последний символ диапазона. В диапазон входят все символы кода ASCII, коды которых не меньше кода первого символа и не больше кода последнего. Например, выражение **[0-9a-fA-F]** задаст любую из шестнадцатеричных цифр, а **[^a-z]** - любой из символов, не являющихся строчной латинской буквой. Внутри квадратных скобок все символы, в том числе **.** (точка), **^**, **\$**, *****, **+**, представляют сами себя. Специальным образом трактуются только символ **^**, следующий сразу за открывающей скобкой, а также символы **-** и ****. Так, **[-+]** представляет "плюс или минус".

В заключение рассмотрения регулярных выражений GREP приведем более развернутый пример. Выражение

```
[+~" "][0-9 ]+\.[0-9 ]*E[+-][0-9 ]+
```

соответствует любому вещественному числу, записанному в форме с плавающей запятой:

[знак] целое . [целое] Е знак целое
без пробелов между составляющими частями.

Теперь об опциях утилиты **GREP**. Они задаются в командной строке вызова **GREP** перед искомой строкой. Каждая опция начинается со знака '-' (минус). После минуса может следовать однобуквенное имя опции, или сразу несколько имен (например, -i -г и -ig задают те же самые две опции). Для отключения опции, включенной по умолчанию, используется знак минус, записываемый вслед за ее именем: так, -г отключает трактовку искомой строки как регулярного выражения. В случае наличия противоречивых установок опций в одной командной строке значимой считается последняя из них. Перечислим опции **GREP**:

-с. Найденные строки на экран не выводятся, вместо этого **GREP** сообщает общее число найденных строк и число найденных строк для каждого файла, в котором производился поиск, в отдельности.

-d. Для каждой из спецификаций файлов, указанных в командной строке, **GREP** производит поиск не только в директории, заданном в этой спецификации (так делается по умолчанию), но и во всех его поддиректориях.

-i. При сравнении строк игнорируется различие между маленькими и большими буквами (это относится, конечно, только к латинским буквам).

-l. Сообщать только имена файлов, содержащих подходящие строки. Как только строка найдена, выводится имя содержащего файла и начинается поиск в следующем файле.

-п. Каждая найденная строка выводится вместе с ее номером в файле.

-о. Формат вывода системы **UNIX**: каждой выводимой строке предшествует имя содержащего ее файла.

-г. Искомая строка трактуется как регулярное выражение (см. выше).

-и. Запомнить опции. Программа **GREP** записывает заданные ей опции в файл **GREP.COM** (то есть в саму себя), при следующем вызове эти опции будут применены по умолчанию. Значения по умолчанию опций, не упомянутых в командной строке, остаются без изменения.

-v. Печатаются только строки, не содержащие искомой.

-w. Поиск слов. Текст, соответствующий регулярному выражению или простой строке поиска, считается подходящим, только если он с двух сторон окружен разделителями. Под разделителями по умолчанию понимаются все символы, кроме латинских букв, цифр и подчеркивания ('_'). Эту установку можно изменить, набрав после опции w в квадратных скобках последовательность символов, которые можно считать допустимыми в словах, то есть не разделителями. Здесь можно использовать диапазоны, как в регулярных выражениях, например, опция -w[a -яА-Я] объявляет элементами слов все буквы кириллицы. Если использовать опцию -w совместно с -и, такая установка будет запомнена как используемая по умолчанию. В случае использования опции -w необязательно даже задавать строку и файлы поиска: достаточно набрать нужные опции, не забыв среди них указать -w.

-z. Расширенная информация. **GREP** печатает имена всех файлов, в которых идет поиск, каждая найденная строка предваряется ее номером, для каждого файла выводится число найденных в нем подходящих строк (если таковые были).

Наконец, третья составная часть параметров **GREP** - спецификации файлов, в которых производится поиск. Можно задавать любое число спе-

цификаций. Каждая из них может быть как обычным именем файла, так и маской (именем файла с использованием метасимволов ? и *). Если маска содержит имя директория, поиск подходящих файлов будет производиться только в этом директории, если имя директория опущено, то в текущем. Опция -d позволяет искать файлы не только в заданном директории, но и во всех его поддиректориях. Если не задано ни одной спецификации файла, GREP может взять текст для поиска строк из стандартного ввода ДОС, при этом на стандартный ввод должен быть назначен какой-либо файл средствами перенаправления (символ < перед именем файла) или с помощью "трубы" (символ | между двумя командами обозначает замыкание стандартный вывод первой из них на стандартный ввод второй).

8.5. OBJXREF.

Как может заключить по названию искушенный читатель, OBJXREF - утилита составления таблицы перекрестных ссылок для OBJ-файлов. Именно так и есть, можно лишь добавить, что кроме привычных таблиц перекрестных ссылок (символ такой-то, определен там-то, используется там-то) она способна составлять отчеты еще семи различных видов, как для отдельного OBJ-файла, так и для набора файлов, причем способы задания этого набора весьма разнообразны. Наравне с OBJ-файлами обрабатываются также и библиотеки объектных программ. Синтаксис вызова OBJXREF традиционен:

OBJXREF [опции] имяфайла [имяфайла ...] .

Каждая из опций начинается с символа \, вслед за ним записывается одно- или двухбуквенное название опции. Как и TLINK, OBJXREF допускает задание файла-подстановки, в котором перечислены все файлы, если их много. Делается это точно так же: символ @ и вслед за ним - имя файла, содержимое которого будет подставлено в командную строку. Возможно использование в одной командной строке нескольких файлов-подстановок, а также файлы-подстановки и имена файлов в любой комбинации. Есть и еще два способа задания набора файлов, они описаны ниже в связи с соответствующими опциями /L и /P.

Программа OBJXREF помещает всю выходную информацию в файл стандартного вывода, то есть обычно на экран. Конечно, при достаточно больших размерах таблиц имеет смысл перенаправлять стандартный вывод в файл, например, так:

OBJXREF /RV *.OBJ >XREF

Если вывод перенаправлен в файл, на экран все же поступает информация о ходе работы и все предупреждения.

Опции OBJXREF:

/Одиректорий. Задает имя директория, в котором размещены обрабатываемые файлы.

/Имяфайла. Набор файлов берется из заданного файла-подстановки, который составлен по правилам подстановок TLINK (список OBJ-файлов, имя EXE-файла, имя MAP-файла, список библиотек - через запятую, подробнее см. предыдущую главу - там, где опции TLINK). Имена EXE- и MAP-файлов, упомянутые в файле-подстановке, игнорируются.

/Имяфайла. Задает ввод набора файлов из файла проекта (см. главу 5 - "Управление проектом"). Расширение в названии файла можно не

указывать, подразумевается .PRJ. При обработке файла проекта имена без расширений и с расширениями .C заменяются на имена с расширениями .OBJ. Зависимость игнорируются. Необходимо помнить, что список файлов проекта сам по себе не задает полную программу: к нему нужно добавить файл C0x.OBJ и стандартные библиотеки Турбо Си (MATHx.LIB, EMU.LIB или F87.LIB, Sx.LIB, возможно, GRAPHICS.LIB), причем указать директорию, в которых они находятся (все это можно сделать в командной строке). Если OBJ-файлы проекта находятся не в текущем директории, дополнительно нужно использовать опцию /O.

/Nимяфайла. Если заданный вами набор файлов достаточно велик, а в действительности вам нужна информация только об одном модуле, опция /N, вслед за которой записано имя этого модуля, блокирует выдачу информации, не относящейся к нему.

/I. В глобальных именах, использующихся OBJ-файлами, при задании этой опции не делается различие между маленькими и большими буквами. Может пригодиться при составлении отчета с использованием разделов библиотек, которые были созданы с опцией /C (см. описание утилиты TLIB ниже в этой главе), или для программ, часть модулей которых создана другим компилятором, различия между этими буквами не делающим.

/F. Библиотеки, указанные в командной строке, обычно включаются не полностью, из них выбираются только те разделы, на которые есть ссылки из OBJ-файлов или из других уже выбранных разделов. Опция /F задает безусловное включение всех разделов библиотек вне зависимости от того, есть ли на них ссылки. Эта опция может оказаться полезной для составления отчета по отдельно взятой библиотеке.

/V. По ходу работы OBJXREF сообщает о своих действиях фразами типа "Считываю файлы", "Связываю модули", "Сортирую сообщения". При задании опции /V информация о ходе выполнения становится подробнее, в частности, программа сообщает о том, какой файл читается в данный момент, и тому подобное.

/Z. Эту опцию можно использовать, если вам почему-либо хочется в отчете по сегментам (см. ниже) получить информацию о сегментах нулевой длины (такие сегменты могут генерироваться компилятором, например, в случае, если в файле не описано ни одной статической переменной).

Группа опций, начинающихся с /R, задает различные форматы отчетов, составляемых OBJXREF.

/RR. Перекрестные ссылки: отчет представляет собой список внешних имен, для каждого из них указан модуль, в котором оно определено, и перечислены модули, на него ссылающиеся. Имена в списке упорядочены по алфавиту.

/RP. Алфавитный список внешних имен; для каждого из них приведено имя модуля, определяющего это имя.

/RM. Список модулей, для каждого из которых указаны внешние имена, им определяемые.

/RX. Список модулей с указанием для каждого из них используемых внешних имен.

/RS. Список имен сегментов, для каждого из которых указаны модули, определяющие сегменты с такими именами, и размер каждого из сегментов (в десятичном и шестнадцатеричном виде). Сегменты, не содер-

жащие инициализированных участков (переменных, которым присвоены начальные значения), отмечаются словом **uninitialized**.

/RC. Список классов (**BSS**, **CODE**, **DATA**), для каждого из которых приведен список всех сегментов данного класса, где указываются размер сегмента и модуль, его определяющий.

/RU. Список неиспользуемых внешних имен. **OBJXREF** выводит список внешних имен, на которые нет ссылок из других модулей. Это ценная информация, особенно для педантов (похоже, педантизм в программировании так же полезен, как мытье рук перед едой: можно обойтись и без него, но в этом случае повышается риск неприятных последствий). Есть два типа внешних имен, которые могут попасть в неиспользуемые. Во-первых, это те, что используются только в том же модуле, где определены. Конечно, разумнее сделать их статическими - неровен час, в другом модуле по ошибке сошлется на это имя. Второй тип - те имена, которые не использует вообще никто. Их лучше всего безжалостно искоренять, хотя бы из тех соображений, что удалять лишние строки программы гораздо легче и приятнее, чем их сочинять.

/RV. Суперотчет, в который включены все 7 видов отчетов (по одному на каждую из только что перечисленных опций).

8.6. TCCONFIG.

Те, кто работают попеременно то в интегрированной среде Турбо Си, то с автономным компилятором, понимают, как важно поддерживать соответствие между установками меню среды, касающимися управления компиляцией, и опциями, задаваемыми ТСС. Почти наверняка и все остальные пользователи Турбо Си согласятся, что такое соответствие по меньшей мере желательно. Как было выяснено в предыдущей главе, набирать вручную все нужные опции не обязательно: достаточно создать файл конфигурации автономного компилятора, где и перечислить все эти опции. Компилятор ТСС ищет файл конфигурации в текущем директории, если не находит - в директории, где расположен он сам. Опции, задаваемые в командной строке, имеют приоритет над опциями файла конфигурации, так что пользователю не нужно менять файл для одноразовой смены задания опций. Но как поддерживать соответствие между меню среды и **TURBOC.CFG** (таково обычное название файла конфигурации ТСС)? И как бы вообще сделать так, чтобы файл конфигурации не нужно было набирать, кряхтя в поисках нужных опций в алфавитном их списке? На помощь приходит утилита **TCCONFIG**: она мигом создает по файлу конфигурации среды эквивалентный (с точки зрения опций компилятора) файл конфигурации ТСС и наоборот.

Способ вызова **TCCONFIG** таков:

TCCONFIG имяфайла [имяфайла2]

Первый параметр задает исходный файл конфигурации, второй параметр - файл, который нужно создать. Если исходный файл содержит конфигурацию среды (это **TCCONFIG** определяет по содержимому файла), файл-результат будет содержать опции ТСС, реализующие те же установки параметров компиляции. Если имя второго файла опущено, ему присваивается стандартное название **TURBOC.CFG**. Если, наоборот, первый файл - файл конфигурации автономного компилятора, результатом будет файл

конфигурации среды (название по умолчанию - TCCONFIG.TC). Предупреждаем: при таком построении файла конфигурации среды теряются пользовательские установки меню среды "Options/Linker" и "Options/Environment", все позиции этих двух меню принимают стандартные значения.

8.7. TCINST.

Любому известно: хорошо, когда инструмент удобно лежит в руке, выкрашен в приятный цвет, не скрипит, и спусковой крючок его нажимается легко. Это высказывание вполне можно отнести и к инструментам программиста-практика. Процесс подгонки инструмента под руку, вкусы и личные пристрастия пользователя по-английски называется customization. В русском языке аналога этому слову не найти, видимо, из-за отсутствия в повседневной жизни народа-языкотворца соответствующего понятия. Система Турбо Си, будучи рассчитана на употребление не только в русскоязычных местностях, имеет свое средство ...эээ... кастомизации. Это средство - программа TCINST - позволяет устанавливать используемые по умолчанию параметры интегрированной среды Турбо Си (такие, как режимы компиляции, директории для поиска файлов и так далее), цвета различных элементов среды на экране, размеры окон, команды редактора и кое-что еще.

TCINST очень проста в использовании: это программа, ведущая диалог с пользователем при помощи системы меню, очень похожих на меню самой интегрированной среды. Все установки, заданные пользователем в сеансе работы TCINST, записываются прямо в файл с программой среды, обычно называемый TC.EXE. Размножив эти файлы и задав в каждом свой набор установок, можно получить по отдельной среде для каждого из пользователей персонального компьютера.

В командной строке вызова TCINST указывается имя EXE-файла, содержащего программу среды Турбо Си, например, так:

```
TCINST D:\USER\SHISHKIN\TCVOLK.EXE
```

Единственная опция TCINST, которую можно задавать перед именем файла, говорит о том, в каком режиме экрана - цветном или черно-белом - будет вестись работа. Обычно TCINST самостоятельно определяет тип адаптера и устанавливает наиболее подходящий по его мнению режим. Опции /b (черно-белый) или /c (цветной) могут все-таки понадобиться, если используется какое-нибудь экзотическое сочетание адаптер/ дисплей, например, цветной адаптер с монохромным дисплеем.

Сразу в начале работы TCINST предлагает пользователю меню из 8 групп установок и одной команды (и команда эта - "Quit"). Часть групп соответствует меню среды Турбо Си "Compile", "Project", "Options", "Debug" - их установки в основном касаются значений отдельных позиций этих меню, принимаемых по умолчанию (то есть при входе в среду Турбо Си, если не используется никакой файл конфигурации). Кроме них основное меню TCINST предлагает отдельные возможности для управления набором команд встроенного редактора, определения режима работы дисплея, установки цветов отдельных частей экрана среды Турбо Си, изменения размеров окон среды и, наконец, для выхода из TCINST с записью внесенных изменений или без таковой. Рассмотрим все позиции основного меню TCINST.

"Compile". Единственная возможность в этом меню - установка имени основного Си-файла **"Primary C file"**, которое будет использоваться по умолчанию. Если вы достаточно долгое время намерены заниматься разработкой единственной программы, которая к тому же состоит из единственного модуля, вы можете найти эту возможность полезной.

"Project". В этом меню четыре позиции, соответствующие первым четырём позициям меню **"Project"** среды: **"Project name"**, **"Break make on"**, **"Auto dependencies"** и **"Clear project"**. Первые три задают значения по умолчанию, последнее используется для очистки позиции **"Project name"**.

"Options". Соответствует одноимённому меню среды. Основное содержание - установки значений различных параметров компилятора, линкера, среды, также директориев для поиска файлов и аргументов, передаваемых программе при запуске. Меню **"Compiler"** и **"Linker"**, а также **"Directories"**, **"Arguments"** и большая часть **"Environment"** полностью соответствуют среде, так что рассматривать их и не будем. Единственное, что может здесь представлять интерес, одна позиция меню **"Environment"**, которая в среде отсутствует. Речь идет о **"Full graphics save"**. Если здесь задать **"On"**, при загрузке среды Турбо Си будет резервироваться 8К оперативной памяти для возможного сохранения графических экранов (сохранение требуется для переключения от среды к пользовательскому экрану и обратно, если программа использует графические возможности). Если вы не собираетесь работать с графикой и жалеете эти 8К, установите **"Full graphics save"** в **"Off"**.

Меню **"Options/Environment"** содержит кроме уже упомянутых еще одну позицию: **"Options for editor"**. Здесь собраны начальные установки режимов редактора (**"Insert"**, **"Indent"**, **"Tab"**, **"Fill"**, **"Unindent"**, **"Tab size"**), а также две неизвестных по среде позиции: **"Editor buffer size"**, позволяющая задавать максимальный размер (в байтах) редактируемого файла, и **"Make use of EMS memory"** - использование расширенной памяти, которую есть смысл устанавливать в **"On"** только на компьютерах с расширенной памятью и наличием драйвера EMS.

"Directories". Это меню полностью соответствует такому же меню среды; здесь задаются начальные значения каждой из позиций.

"Arguments". Здесь задаются аргументы запускаемых программ, те самые, что в среде записываются в **"Options/Arguments"**.

"Debug". Здесь только две позиции, устанавливающие начальные значения для переключателей **"Source debugging"** и **"Display swapping"**.

"Editor commands". Пожалуй, наиболее интересная из всех возможностей, предоставляемых TCINST. Здесь для любой команды встроенного редактора среды Турбо Си можно задать нажатия клавиш, эту команду вызывающие. Договоримся, кстати, здесь "клавишами" называть нажатия любой сложности (из одной или двух клавиш, или даже типа **<CtrlQ><CtrlI><ShiftF4>**) - ради краткости и выразительности изложения. Процесс задания клавиш выглядит следующим образом: при выборе из основного меню TCINST позиции **"Editor commands"** на экране появляется список команд, оформленный в виде трех столбцов: в первом дается название команды (например, **"New line"**, **"Top of screen"**, **"End of file"**), во втором и третьем - обозначения клавиш, осуществляющих соответствующую команду (скажем, для команды **"Cursor left"** - **"Курсор влево"** - во втором столбце записано **<CtrlS>**, в третьем - **<Lft>**, последнее обозначает клавишу "стрелка влево"). Второй столбец содержит так называемые первичные (primary)

клавиши, третий - вторичные (secondary). Первичные и вторичные клавиши одной и той же команды являются синонимами (как <CtrlD> и <Rgt>), причем, как ни странно, вторичные клавиши "главнее" первичных, то есть, если одна и та же клавиша задана как первичная у одной команды и как вторичная у другой, реально она будет работать как клавиша той команды, для которой она - вторичная.

Чтобы изменить назначение клавиши команде, нужно подвести подсветку к строке команды, выбрать (подсветкой же) вторичную или первичную клавишу и нажать Enter. Откроется окошко, в котором вы можете набрать обозначение клавиши, вызывающей данную команду. Ввод обозначений довольно хитер: есть три режима обработки нажатий (их названия светятся в нижней части экрана TCINST) - "WordStar-like", "Ignore case" и "Verbatim". Каждый из этих способов по-своему задает реакцию на нажатия управляющих клавиш в редакторе и по-своему воспринимает задание клавиш в TCINST. Общее у них то, что первая клавиша команды не может быть алфавитно-цифровой, то есть обязана быть управляющим символом (код от 0 до 31, набирается с использованием клавиши Ctrl) либо одной из специальных клавиш (F2-F12, различные комбинации клавиш Alt и букв/цифр, стрелки и так далее). В чем разница между режимами, станет понятно из их описания, приводимого ниже.

"WordStar-like". Как в редакторе WordStar. Нажатие любой буквы или одного из символов '[', ']', '\', '^', '-' интерпретируется как управляющая комбинация из Ctrl и нажатого символа. Так, Z, z и Ctrl-Z здесь эквивалентны; команда, например, <Ctrl-R><Ctrl-Z> может быть набрана в редакторе как Ctrl-R Ctrl-Z, как Ctrl-R Z и как Ctrl-R z - все с одинаковым результатом.

"Ignore case". Без различия регистра. Отличается от режима "WordStar-like" тем, что делается различие между клавишами, нажатыми одновременно с Ctrl, и без нее. Проще всего показать это на примере: Ctrl-R Z есть то же самое, что Ctrl -R z, но не то же, что Ctrl-R Ctrl-Z.

"Verbatim". Буквально. Здесь все три варианта нажатий буквенных клавиш различны: Ctrl-R Ctrl-Z, Ctrl-R Z и Ctrl-R z.

При наборе команд особый смысл имеют клавиши Backspace (удаление последнего элемента команды), Enter (конец набора), Esc (отказ от изменения команды), F2 (восстановить предыдущую установку команды), F3 (счистить команду) и F4 (переключатель режимов ввода). Если хочется набрать одну из этих клавиш F2, F3, F4 как часть команды, перед ее нажатием вводят "'" (обратная кавычка - не путайте с апострофом). Для ввода остальных спецклавиш используется Ctrl:

Ctrl-H соответствует Backspace,

Ctrl-[соответствует Esc,

Ctrl-M соответствует Enter.

Возможности по заданию клавиш для команд редактора богатейшие. Есть, конечно, и ограничения, но их совсем мало. Как уже говорилось, в качестве первого символа команды нельзя использовать буквы, цифры и знаки пунктуации. Кроме того, в командах нельзя использовать F1 и Alt-F1 (ибо не останется способа вызвать систему помощи). Длина команды - не более шести нажатий, причем некоторые клавиши соответствуют двум или даже более нажатиям (комбинации с использованием Alt, стрелки, функциональные клавиши и так далее).

"Display mode". Режим дисплея. Обычно среда Турбо Си перед началом работы разбирается с видеорежимом, в котором работает компьютер в момент ее вызова, и под него подстраивается. Может, однако, возникнуть ситуация, когда этот интеллект среды покажется вам ненужным: например, обычно вы работаете в режиме, в котором Турбо Си работать не может, или просто обладаете такой хитрой аппаратурой, что среда ошибается в ее идентификации. Меню **"Display mode"**, которое придет вам на помощь, содержит 4 позиции:

"Default". По умолчанию. Обычный случай: среда работает в том режиме, в котором вызвана.

"Color". Цветной. Всегда устанавливается 80-символьный цветной режим, если обнаружен цветной адаптер. По выходе из среды восстанавливается режим, из которого она была вызвана.

"Black and white". Черно-белый. Даже с цветным адаптером будет включен черно-белый режим (80 символов в строке). После выхода также восстанавливается предыдущий видеорежим.

"Monochrome". Одноцветный. Для одноцветных адаптеров (которые изображают различные цвета оттенками одного цвета) будет устанавливаться одноцветный режим, даже если среда вызвана из черно-белого режима.

Для цветных адаптеров при быстром выводе на экран может возникать характерный "снег". Чтобы узнать, обладает ли этим свойством используемый вами адаптер, TCINST выполняет видеотест. Выбрав один из режимов **"Default"**, **"Color"** или **"Black and White"**, вы сможете посмотреть на экран и ответить, видели ли вы "снег". Ответ выбирается из трех вариантов:

- да, "снег" был. Среда станет работать с экраном чуть медленнее (трудно сказать, ощутите ли вы замедление), но "снега" не будет;

- нет, "снега" не было. Быстрый вывод на экран будет работать без проверок на "снег" и без проблем (впрочем, проблем точно не будет только на вашем или таком же компьютере);

- был "снег" или не было, перед началом работы всегда проверять аппаратуру. Этот уклончивый ответ даст системе понять, что вы носите данный экземпляр Турбо Си по разным машинам, или достаточно часто меняете адаптер в своем компьютере.

"Set colors". Установить цвета. Красивая игрушка для тех, кому надоело трудиться, а употреблять настоящие игры не позволяет шеф. Экран Турбо Си содержит полсотни элементов, каждый из которых выводится приписанным ему цветом. Эти цвета записаны в EXE-файле среды, и их можно изменять при помощи TCINST. Помимо "ручной" установки цветов (позиция **"Customize colors"**) можно выбрать один из трех готовых наборов цветов - предопределенный (**"Default"**), "бирюзовый" (**"Turquoise"**) и "фиолетовый" (**"Magenta"**). Поговорить об этих наборах можно было бы, но лучше на них посмотреть - попробуйте!

Ручная установка цветов различных элементов экрана устроена довольно остроумно и удобно. Сначала при помощи меню двух уровней пользователь выбирает интересующий его элемент (например, **"Directory Box"** в первом меню и **"Border"** во втором, если хочется поменять цвет рамки окна, в котором выводится список имен файлов). Затем на экране появляется окно, в котором своими цветами выделены различные элементы, среди них - и выбранный. В левой части дисплея - карта цветов, в которой изображены все возможные комбинации цветов текст/фон для текущего видеорежима (запущенного в TC.EXE при помощи меню **"Mode for display"**).

Карта содержит "курсор" (рамку), гоняя которую по карте, можно выбрать цвет по душе: цвет элемента одновременно показывается в правом окне.

"Resize windows". *Изменить размер окон.* В двухоконном режиме (редактор/Watch или редактор/Message) нижнее окно по мере наполнения способно расти, но не более, чем до определенного предела. Предел этот и устанавливается командой "Resize windows". Делается это очень просто: выбрав эту команду и получив на экране изображение двух окон, стрелками вверх/вниз двигаете верхний край нижнего окна. Окно не может содержать меньше одной строки - остальное в вашей власти.

"Quit/save". *Выйти/записать.* Установив все, что требуется, выходите. На вопрос "Save changes to TC.EXE?" можно ответить нажатием Y - все установки попадут в программу среды - или N - все останется по-прежнему, как будто TCINST вы не вызывали.

8.8. THELP.

THELP - штука, которую по достоинству оценят все любители командной строки, и не только они. Работая в каком-то другом редакторе (не в среде) или отлаживая программу каким-то другим отладчиком (не встроенным), можно иметь доступ к системе помощи по Турбо Си, не уступающей по богатству текстов аналогичной системе сред (точнее, ей эквивалентной - обе они используют один и тот же файл текстов помощи), а по сервису далеко ее превосходящей.

THELP - резидентная программа (кстати, очень небольшая - всего 8K), как и все резидентные программы, загружать ее нужно до того, как вызывается редактор или отладчик, в котором вы собираетесь ее использовать.

О запуске и возможных опциях поговорим чуть позже, а сейчас разберемся с тем, как ведет себя THELP в работе.

Будучи загруженной, THELP вызывается нажатием определенной клавиши. Если не установлено другой клавиши, то это будет 5 на цифровой клавиатуре. Другую клавишу можно установить в момент загрузки THELP при помощи опции /K или переопределить "на ходу", послав опцию /K при помощи /R (подробности ниже). Когда нажата клавиша, вызывающая THELP, читается слово, находящееся на экране под курсором. Если THELP есть что сказать об этом слове, выводится соответствующий ему текст. Если нет - оглавление системы помощи; короче говоря, здесь THELP ведет себя точно так же, как среда Турбо Си при нажатии Ctrl-F1.

Внутри THELP устроен очень похоже на систему помощи среды. Каждый текст помощи связан с родственными ему текстами ссылками, по которым ходят стрелками и выбирают нажатием Enter. Страницы длинных текстов листаются PgUp/PgDn, F1 выдает оглавление, Alt-F1 - предыдущий текст, а Esc заканчивает сеанс. Кроме этого есть еще несколько команд, отсутствующих в системе помощи среды, но тем не менее весьма полезных.

Каждая из них вызывается нажатием одной из клавиш латинской буквенной клавиатуры, причем несущественно, в каком регистре.

F. Сменить help-файл. THELP способен работать как с файлом помощи по Турбо Си, так и с файлом помощи по Турбо Паскалю (или с любым другим файлом, составленным в том же формате каким-нибудь умельцем). Если вы в процессе работы переключаетесь с Си на Паскаль или обратно,

достаточно после вызова THELP нажать F и в ответ на приглашение набрать имя нужного help-файла. Нужно сказать, что совместимы только help-файлы Турбо Си версии 2.0 и Турбо Паскаля версии 5.0. Аналогичные файлы Турбо Си++ и Турбо Паскаля версии 5.5 (там тоже есть программы THELP) имеют другой формат.

I. То слово на экране THELP, которое в данный момент подсвечено (ключевое слово, стандартный идентификатор, команда THELP и т. п.), вставляется в буфер клавиатуры, после чего THELP заканчивает сеанс. В итоге ситуация выглядит так, будто вы вышли из THELP и набрали на клавиатуре этот текст. Если вы находитесь в редакторе, он попадет в текст редактируемого файла, если в среде ДОС - в командную строку, и так далее.

J. Переход к странице по номеру. В отличие от системы помощи среды Турбо Си, в TCHELP все "страницы" текстов пронумерованы. Если вы настолько опытный в использовании THELP, что помните номера интересных вам страниц, то вас остается только поздравить - знаниями своими вы можете пользоваться.

K. Поиск слова. Предлагается ввести слово - идентификатор, ключевое слово Турбо Си или любое другое, для которого есть свой текст помощи. Текст этот находится и выводится на экран.

P. Похоже на команду I, но в буфер клавиатуры вставляется весь текст текущего экрана. Каждая строка экрана заканчивается "нажатием" Enter. Так как передача всего текста занимает достаточно долгое время, ее можно успеть прервать нажатием Ctrl-C или Ctrl-Break.

S. Текст текущего экрана запоминается на диске в текстовом файле THELP.SAV в текущем директории. Если такой файл уже существует, текст добавляется ему в конец.

Наконец, нажатие Ctrl-F1 вызывает на экран список всех команд THELP, только что перечисленных.

Теперь о том, как запускать THELP и какие у него имеются опции. Запуск THELP прост:

THELP опции

Опции THELP начинаются с символа /, вслед за которым записывается однобуквенное имя опции. Некоторые опции имеют параметры, они пишутся вплотную к имени опции. Для опций, являющихся переключателями (включить/выключить), включение обозначается необязательным символом + вслед за именем опции, отключение - минусом.

/B. При включении этой опции THELP будет производить вывод на экран при помощи прерываний BIOS (обычно же это делается прямой записью в видеопамять). Разница - в скорости, но и в гарантированном отсутствии "снега" на экране.

/Sxxx. Установка цветов выводимых THELP изображений. Следом за названием опции S задаются три шестнадцатеричные цифры, определяющие элемент окна THELP, цвет фона и цвет текста соответственно. Номера элементов окна:

1 и 2 - обычный текст; 3 и 4 - ссылки на другие тексты; 5 и 6 - рамка; 7 и 8 - подсвеченная ссылка на другой текст.

В каждой паре нечетная цифра относится к цветному режиму, четная - к одноцветному. В цветном режиме цвета текста и фона задаются стандартным способом: фон - цифра от 0 до 7 (0 - черный, 1 - синий, 2 - зеленый, 3 - голубой, 4 - красный, 5 - фиолетовый, 6 - коричневый, 7 - серый), текст - от 0 до F, причем цвета с номерами больше 7 имеют

повышенную яркость. Для получения мерцания к цвету фона нужно добавить 8. С монохромным режимом все сложнее, стандарта здесь нет, так что придется поэкспериментировать.

/Dдиректорий. Задает имя директория, в котором THELP будет размещать свои рабочие файлы. Файлы эти носят имена THELP.SW1 и THELP.SW2, их можно без страха удалять по окончании работы с THELP. Если опция /D не задана, файлы будут располагаться в том же директории, что сам THELP (в версиях ДОС от 3.0 и выше), или в корневом директории диска C (версии ДОС до 3.0).

/Гимяфайла. Задает полное (подчеркиваем: полное, с дисководом и маршрутом) имя help-файла, который будет использоваться. По умолчанию используется файл TCHELP.TCH из того же директория, где расположен THELP.

/Kxxxx. Задает клавишу, вызывающую THELP. В опции задаются две пары шестнадцатеричных цифр, первая из них - состояние клавиш Shift, Ctrl и Alt, вторая - скан-код клавиши. Тем, кто все понял, дальнейшие объяснения не требуются. Остальным рекомендуется почитать что-нибудь об устройстве IBM PC, а именно о том, как обрабатываются нажатия клавиш.

/Lxx. Число строк на экране. Задается одно из чисел 25, 43 или 50 - в зависимости от того, в каком режиме вы собираетесь работать. Обычно THELP определяет число строк на экране самостоятельно, но на некоторых компьютерах ошибается. Если задана опция /L, проверяться ничего не будет, а THELP будет считать, что экран содержит всегда столько строк, сколько сказано.

/M. Опция, которая вряд ли пригодится большинству из читателей. Речь идет о работе с двумя мониторами. Один из них - цветной - используется как основной, другой - монохромный - как вспомогательный. Если задана опция /M, тексты THELP будут направляться на вспомогательный экран и не мешать работе с основным. Отметим, что опция /M не действует, если задана опция /B.

/Px. Скорость передачи текста в буфер клавиатуры (которая производится по командам I и P). Задается число: 0 - медленная, 1 - средняя, 2 - быстрая. Замедлить заполнение буфера бывает нужно при работе с задумчивыми редакторами, если они не успевают вынимать символы из буфера с той же скоростью, с какой они туда поступают. Также могут помешать быстрому считыванию некоторые резидентные программы.

/R. Очень интересная возможность: загрузив THELP, вы можете потом дополнительно передать ей любые опции. Делается это так: если в командной строке есть опция /R, все остальные опции передаются резидентной части THELP, изменяя более ранние установки. Единственная опция, которая не может быть передана таким образом, - /S, о ней ниже.

/Sx. Расположение рабочих файлов. Задается цифра: 1 - файлы располагаются на диске, 2 - в расширенной памяти (EMS), 3 - в оперативной памяти. В последнем случае на диске рабочие файлы THELP.SW1 и THELP.SW2 не создаются, но вместо 8K памяти THELP начинает занимать 40K.

/U. Удаляет THELP из памяти. Не забудьте предварительно удалить все резидентные программы, которые вы загружали после запуска THELP.

/W. Подобно некоторым другим утилитам Турбо Си, THELP способна запоминать опции, заданные в командной строке, в собственном COM-фай-

ле. Таким образом, если вы задаете в командной строке опцию /W, все остальные опции (кроме /R) запоминаются и будут использоваться по умолчанию при последующих запусках THELP.

8.9. TLIB.

Назначение программы TLIB - обслуживание библиотек объектных программ. Это могут быть как пользовательские библиотеки, так и стандартные (если у вас есть желание и возможность их изменять).

Прежде, чем заняться рассмотрением возможностей TLIB, поговорим немного о том, зачем вообще нужны библиотеки и чем выгодно их использование. Строго говоря, библиотеки объектных программ - не обязательный для создания программы элемент. Библиотека логически представляет собой просто набор OBJ-файлов (будем в дальнейшем называть их объектными модулями или просто модулями), оформленный в виде одного-единственного файла чуть более сложной структуры, позволяющей читать отдельные модули, заменять их, удалять и так далее. Это значит, что любая библиотека может быть заменена соответствующим числом отдельных OBJ-файлов, и результат сборки выполняемой программы линкером будет при этом тем же самым.

В чем же разница? Во-первых, в использовании дискового пространства. Библиотека занимает меньше места на диске, чем входящие в нее OBJ-файлы, размещенные по отдельности. Во-вторых, в скорости сборки программы. Линкеру нужно открыть и закрыть каждый из OBJ-файлов, которые он включает в программу (открытие - процесс довольно долгий, и прибавка во времени сборки может быть существенной). В случае использования библиотеки открыть нужно только один файл. Если используется возможность создания в библиотеке расширенного словаря (ознакомьтесь с опцией /E, описанной ниже), то скорость сборки повысится дополнительно. Но эти соображения - количественные, в конце концов на сверхбыстром компьютере с резиновым винчестером разница может и не ощущаться. Есть и другой повод использовать библиотеки, относящийся к области технологий.

Начнем с того, что сообщим об одной неприятной особенности линкера Турбо Си (как встроенного, так и автономного), отличающей его в худшую сторону от аналогичных инструментов Турбо Паскаля или Турбо Си++. Если в программе используется глобальная функция из какого-то модуля, в EXE-файл этой программы включен будет весь этот модуль, даже если 99% его кода никогда не используется. В итоге, если вы создали, скажем, большой пакет для обработки чего-нибудь и поставляете его пользователю в виде исходного текста или OBJ-файла, в один прекрасный момент вы рискуете обнаружить, что пакетом не пользуются: слишком велика плата за вызов одной-двух функций из него. Естественный выход - разбить пакет на множество как можно более мелких модулей. Но, помимо уже упомянутого проигрыша в дисковом пространстве и скорости сборки, с таким набором просто очень трудно работать: в проект нужно включать кучу дополнительных файлов (каждый раз приходится вычислять, каких именно), и удовольствие пользователей от работы с таким пакетом вряд ли увеличится.

Выход из этого затруднения состоит как раз в использовании библиотеки. Вы разбиваете пакет на множество мелких модулей, транслируете их, составляете из полученных объектных файлов библиотеку, и проблемы как рукой сняло: пользователь имеет дело только с двумя файлами (библиотекой и хэдером, описывающим ее функции и переменные), в то же время в его выполняемую программу попадает ровно столько программного кода, сколько ему нужно для работы (конечно, если автор программы постарался хорошо распределить функции по файлам). Еще раз добавим, что повысилась скорость сборки и снизились затраты дискового пространства.

Из всего вышесказанного можно понять, какого рода пользователи Турбо Си заинтересуются вопросом создания и обслуживания библиотек объектных модулей. Это те, кто готовит достаточно богатые наборы функций, предназначенных для использования в составе различных проектов (примерами могут служить, скажем, оконные системы, средства обслуживания файлов нетривиальной организации, пакеты для работы с динамической памятью - список можно продолжать бесконечно). В то же время авторы автономных программ, выдающие на-гора лишь EXE-файлы, могут о библиотеках (точнее, об их создании) особенно не беспокоиться, хотя для общего развития и им советую ознакомиться с программой TLIB.

TLIB умеет:

- создавать библиотеки;
- добавлять OBJ-файл в библиотеку в виде нового модуля;
- удалять модуль из библиотеки;
- заменять модуль новым OBJ-файлом;
- извлекать из библиотеки модуль и делать из него OBJ-файл (при этом модуль может быть удален из библиотеки).

Помимо всего этого можно вывести в файл или на экран (принтер) список всех модулей библиотеки, задать чтение параметров из файла (файл подстановки, как и у некоторых других утилит Турбо Си). А еще TLIB имеет 2 опции. Начнем с общего вида вызова TLIB. (как обычно, квадратные скобки обозначают необязательные компоненты).

TLIB библиотека [/C] [/E] [команды] [,листинг]

“Библиотека” - это имя файла, содержащего библиотеку, с которой вы желаете работать. По умолчанию принимается расширение .LIB. Если указанного вами файла не существует, при необходимости (то есть перед выполнением команды добавления модуля) он будет создан.

“Листинг” - тоже имя файла. Если он задан (заметьте: в конце списка параметров, через запятую от него), в этот файл помещаются алфавитный список всех модулей библиотеки (кстати, имя модуля есть имя OBJ-файла, из которого он получен), и для каждого модуля - алфавитный список внешних имен, в модуле определенных. Частое применение этой возможности - составление файла с составом библиотеки без других действий, выглядит это, например, так:

TLIB LIBRA , RY

Список модулей библиотеки LIBRA.LIB будет помещен в файл RY.LST (.LST - расширение, придаваемое файлу листинга по умолчанию).

Основных команд у TLIB три: добавить модуль, удалить модуль, извлечь модуль (записав его на диск в виде OBJ-файла). Записываются они вслед за опциями (если их нет, вслед за именем библиотеки). Каждая команда записывается как одно- или двусимвольное название команды, вслед за которым (возможно, через пробелы) идет имя файла или модуля

- параметра команды. Одну команду от другой пробелами можно не отделять. Вот команды TLIB:

+файл. Добавить модуль в библиотеку. Если модуль с таким именем в библиотеке уже есть, он не изменяется (о чем TLIB сообщает). В имени того файла, из которого получается модуль, можно указывать дисковод и маршрут; по умолчанию предполагается расширение .OBJ.

-модуль. Удалить модуль из библиотеки. *модуль Извлечь модуль из библиотеки. На диске создается файл, содержащий указанный модуль. Директорий, в который помещается файл, можно указывать в имени модуля. Расширение по умолчанию - .OBJ, и неясно, зачем могло бы понадобиться другое.

Этих трех команд в принципе достаточно, чтобы проводить все необходимые манипуляции с набором модулей, составляющим библиотеку. Но, видимо, из человеколюбия, создатели TLIB предусмотрели "двойные" операции:

+модуль или +модуль Заменить модуль. Модуль сначала удаляется из библиотеки (пусть вас не смущает возможность набрать сначала +, а потом -), а затем добавляется - из файла с соответствующим именем.

*модуль или *-модуль Извлечь модуль с удалением. Сначала модуль переписывается на диск в виде файла, после чего удаляется из библиотеки.

Каждая из этих команд, как нетрудно видеть, заменяет собой две "основных" команды с одинаковым именем модуля. Важно понимать, что TLIB относится к двойным командам именно так: сначала он выполняет первую, затем вторую; и если, набрав команду +-TREASURE, Вы не имеете в текущем директории файла TREASURE.OBJ, TLIB преспокойно удалит модуль TREASURE из библиотеки и лишь после того проверит, а есть ли, чем заменять этот модуль. Итог - сокровище потеряно... Похоже, именно из-за возможности таких несчастий TLIB всегда оставляет аварийную копию библиотеки - файл с расширением .BAK.

Говоря в предыдущем абзаце о том, что TLIB выполняет сначала первую часть двойной команды, а следом - вторую, я не раскрыл всей правды и мог быть не совсем правильно понят. Дело в том, что порядок выполнения команд у TLIB довольно своеобразный. Сначала выполняются все команды * из заданных при вызове TLIB (в том числе те, что являются частью двойной команды "извлечь и удалить"), затем - все команды удаления (в том числе из команд +- и *-), и лишь после этого настает черед команд добавления. Отложите этот способ действий где-нибудь в своей памяти и относитесь к возможным его последствиям с пониманием.

Помимо двойных команд, TLIB предлагает еще одно средство сокращения длинных командных строк. Речь идет об уже, возможно, знакомых вам файлах подстановки. В любом месте командной строки вызова TLIB можно поставить символ @ и вслед за ним - имя файла. Дальнейшая часть командной строки будет считана из этого файла, называемого файлом подстановки. Если после параметра @имяфайла в строке есть еще какие-то параметры, к их обработке TLIB перейдет после обнаружения конца файла подстановки. Очевидно, что это позволяет использовать в одной строке несколько файлов подстановки. Содержимое файла подстановки может занимать несколько строк. Признаком продолжения параметров TLIB на следующей строке файла подстановки служит & в конце строки.

Хороший пример ситуации, когда использование файла подстановки позволяет резко повысить производительность труда: вы перекompилировали

все модули своей библиотеки из трех дюжин модулей и хотите всю библиотеку обновить. Руками такую работу не проделать (разве что у вас почасовая оплата труда), решение же писать длинный batch-файл, содержащий 36 строк вида

TLIB 36ONES.LIB -+MODxx

выглядит довольно дурацким. Все проблемы решает файл подстановки. Вы изготавливаете файл, скажем, ADD36.RSP, такого вида:

+ MOD00 &

+ MOD01 &

...

+ MOD34 &

+ MOD35

после чего смело удаляете старую библиотеку и набираете

TLIB 36ONES @ADD36.RSP ,36ONES.LST

Готово! Ситуация эта, надо сказать, в жизни "библиотекарей" нередкая, так что стоит с самого начала построения библиотеки запастись файлом такого рода.

Последнее, о чем можно сказать в связи с утилитой TLIB, - те самые две опции, которые мы упомянули в начале разговора.

/E. Библиотека может содержать структуру, называемую Расширенный Словарь (Extended Dictionary). Словарь этот позволяет линкеру Турбо Си извлекать модули из библиотеки гораздо быстрее, чем обычно. Плата за ускорение невелика: порядка нескольких процентов от размера библиотеки. Чтобы создать Расширенный Словарь в библиотеке, нужно задать опцию /E одновременно с какой-нибудь изменяющей библиотеку командой (+, - или +-). Если вам нечего изменить, употребите в этой команде имя несуществующего модуля - тоже сойдет.

/C. TLIB не позволяет заносить в библиотеку модули, пересекающиеся по определяемым именам - все имена библиотеки должны быть различными. Если вы попытаетесь добавить модуль, в котором определено имя, совпадающее с одним из имен, определенных в уже имеющемся модуле, TLIB даст отказ и новый модуль отринет (кстати, вот еще случай, когда команда +- может привести к удалению модуля). Опция /C влияет на то, какие имена TLIB считает "совпадающими", точнее, как он относится к строчным и прописным буквам. По умолчанию различия между строчными и прописными буквами не делается, и, например, имена file и FILE будут считаться одинаковыми. Лишь задание опции /C приводит к обычной для Си работе: все буквы разные, кроме одинаковых по коду. Это подозрительное решение авторы TLIB объясняют тем, что не все линкеры умеют различать большие и маленькие буквы. Хотя звучит это объяснение не слишком убедительно, сделаем вывод: если вы точно знаете, что составляемая вами библиотека не будет использоваться с линкером, отличным от TLINK, употребляйте опцию /C при добавлении модулей. Если не уверены - воздержитесь, но свобода выбора внешних имен слегка усядет.

8.10. TOUCH.

Маленькая программа, выполняющая тривиальное действие. Получив в качестве параметра одну или несколько спецификаций файлов, она для

всех файлов, удовлетворяющих заданным спецификациям, устанавливает в качестве даты и времени последнего изменения сегодняшнюю (конечно, с точки зрения компьютера) дату и текущее (с той же оговоркой) время. Несмотря на простоту, утилита TOUCH позволяет выпутаться из по крайней мере двух весьма пикантных ситуаций.

Ситуация номер один. Каким-то образом в ваш проект затесался исходный файл, время (о дате ради краткости говорить не будем, но будем это подразумевать) последнего изменения которого больше текущего. Такой мелочи достаточно, чтобы Make (автономный или встроенный - неважно) перетранслировал этот файл при каждом своем вызове: ведь время создания соответствующего OBJ-файла каждый раз будет меньше "времени изменения" исходного текста. Так будет до тех пор, пока текущее время не обгонит время изменения исходного текста. Выход:

TOUCH этотфайл

После еще одной трансляции время записи OBJ-файла наконец станет больше (теперь-то временем последнего изменения Make считает время запуска TOUCH), и проблема исчезнет.

Ситуация номер два. Вы (умышленно или нечаянно) изменили в сторону увеличения дату последней модификации одного из узловых файлов проекта, от которого зависит несколько десятков других файлов. Такое нередко происходит при неосторожном просмотре файла в редакторе (случайно нажал пробел, при выходе увидел вопрос "Сохранять ли изменения?" и от неожиданности сказал "Да"). Следующий вызов Make будет означать массовую перекомпиляцию - хорошо, если на одну-две минуты, а то ведь и на все полчаса! Выход - "тронуть" все файлы, зависящие от псевдоизмененного, перечислив их в командной строке вызова TOUCH. Предупреждение: если вы пользуетесь командой Make интегрированной среды, то вы не избавились от перекомпиляции, а лишь отложили ее до того, когда переключатель "Project/Auto dependencies" примет положение "On" - в этом случае проверяется еще и время, записанное в OBJ-файле.

ЧАСТЬ II. СПРАВОЧНОЕ РУКОВОДСТВО ПО ТУРБО СИ

9. ЛЕКСИКА

В этой части книги мы описываем язык Си для операционного окружения Турбо Си, созданного фирмой Borland для работы на IBM PC в среде MS DOS. Вместе с тем, мы всюду стараемся подчеркнуть отличие стандарта Си (так называемого ANSI Си) от тех дополнений, которые внесены в него фирмой Borland. За немногими исключениями различия объясняются специфическими особенностями аппаратуры IBM PC.

Мы без колебаний неоднократно повторяли одно и то же, если нам казалось, что это облегчит читателю его и без того нелегкий труд.

Язык - это совокупность предложений, каждое из которых удовлетворяет синтаксису этого языка. Предложения представляют собой конечные последовательности слов, которые чаще называют лексемами и которые состоят из последовательности (в формальных грамматиках их называют цепочками) литер, иначе - элементов алфавита.

Алфавит языка Си состоит из всех печатных символов, расположенных на стандартной клавиатуре:

- цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; - латинские прописные и строчные буквы a - z, A - Z; - русские прописные и строчные буквы; - символ подчеркивания (_); - другие знаки (.,):\'/?*."-+&^~!|=;[]{}\$ и т.д.

Существует шесть классов лексем (token): идентификаторы, служебные слова, константы, строки, знаки операций и некоторые другие разделители. Пробелы, символы табуляции, переходы на новую строку и комментарии (все вместе они называются "пустыми символами"), как это будет описано ниже, используются для отделения одной лексемы от другой. Вместо одного из "пустых символов" можно использовать любое их количество.

Если входной текст уже разбит до некоторого символа на лексемы, то в качестве следующей лексемы выбирается максимально длинная последовательность символов, которые могут составить лексему.

9.1. КОММЕНТАРИИ

Пара символов /* начинает комментарий, пара символов */ заканчивает его. В стандартном Си примечания не могут быть вложенными. В Турбо Си имеется возможность разрешить вложенные комментарии (переключатель Nested comments...Он в меню Options/Compiler/Source или оп-

ция командной строки -С). Все же, транспортабельности ради, вложения комментариев следует по возможности избегать. Вот, например, текст, в котором комментарий использован для временного удаления части программы:

```
/* следующий оператор временно удален
   for (i=0; i < BOUND; i++) /* обход массива */ do_smthng(arr[i]);
*/
```

Он, конечно, функционально эквивалентен следующему (по поводу операторов, начинающихся с символа #, см. главу 15):

```
#if 0
   for (i=0; i < BOUND; i++) /* обход массива */ do_smthng(arr[i]);
#endif
```

Разница лишь в том, что второй вариант будет корректен для любого компилятора, чего не скажешь о первом.

Некоторые компиляторы комментарии вообще пропускают, и потому комментариев можно употреблять даже внутри лексемы. В Турбо Си комментарий эквивалентен пробелу.

9.2. ИДЕНТИФИКАТОРЫ

Любой идентификатор - это последовательность латинских букв и цифр, причем первым символом обязательно должна быть буква. Символ подчеркивания ('_') считается буквой. Строчные и прописные буквы в Си считаются различными. В Турбо Си можно использовать в идентификаторах символ '\$' (доллар); все же буквой он не считается и потому не может быть первым символом идентификатора.

Количество "значимых" символов в идентификаторах различно для разных компиляторов. В Турбо Си оно по умолчанию равно 32 (это значит, что идентификаторы, в которых совпадают первые 32 символа, считаются одинаковыми). Имеется возможность заменить эту длину на любую другую от 1 до 32 - использованием либо опции *Identifier length* в меню *Options/Compiler/Source*, либо опции командной строки *-i#*, где # - требуемое число символов.

Аналогично, в глобальных именах, используемых ассемблерами и загрузчиком, значащими считаются 32 первых символа. Имеется возможность управлять тем, считаются ли различными строчные и прописные буквы в глобальных именах во время редактирования внешних связей: опции *Case-sensitive link...* On меню *Options/Linker* или /с редактора связей (программа *TLINK*) задают трактовку строчных и прописных букв как различных. Тем не менее, в глобальных именах, описанных с модификатором *pascal* (специфичным для Турбо Си), различие при редакции связей не делается никогда.

9.3. КОНСТАНТЫ

Здесь перечислены имеющиеся виды констант.

Целые константы.

Любая целая константа, представляющая собой последовательность цифр, считается восьмеричной, если она начинается с цифры нуль ('0'); в противном случае это десятичная константа. В восьмеричные константы не могут входить цифры 8 и 9. Если последовательность цифр начинается с пары 0х или 0Х (за цифрой нуль следует латинская буква 'х' или 'Х'), то константа считается шестнадцатеричной. Набор шестнадцатеричных цифр - это цифры 0...9 и буквы а...f или А...F, представляющие числа 10...15 соответственно. Допустимы любые константы от 0 до 4294967295 (основание 10). Отрицательные десятичные константы компилятор трактует как беззнаковые с предшествующим унарным минусом.

Длинные константы.

Суффикс L (или l), следующий после значения (без пробела между ними), сообщает компилятору о том, что константу следует трактовать как длинное целое независимо от ее значения. Аналогично, суффикс U (или u; эта возможность отсутствует в "библии") делает ее беззнаковой (unsigned), и константа будет иметь тип unsigned long, если значение ее больше 65535, независимо от основания.

Ниже приведена таблица типов констант без суффиксов.

десятичные	
0-32767	int
32767-2147483647	long
2147483647-4294967295	unsigned long
4294967295 и больше	переполнение
восьмеричные	
00-077777	int
0100000-0177777	unsigned int
1000000-0177777777	long
0100000000000-037777777777	unsigned long
037777777777 и больше	переполнение
шестнадцатеричные	
0x0000-0x7FFF	int
0x8000-0xFFFF	unsigned int
0x10000-0xFFFFFFFF	long
0x80000000-0xFFFFFFFF	unsigned long
0x100000000 и больше	переполнение

Переполнение компилятором никак не отмечается, и значением константы будут считаться младшие биты записанного.

Константы с плавающей точкой.

Константа с плавающей точкой строится из: целой части, точки ('.'), дробной части, символа 'е' или 'Е' и целого показателя (экспоненты). Целая и дробная части, а также экспонента - последовательности цифр. Целую или дробную часть можно опустить, но не обе одновременно. Аналогично, десятичная точка или символ 'е' ('Е') с показателем могут отсутствовать, но не то и другое сразу. Константа с плавающей точкой преобразуется к типу `double`. Исключения для Турбо Си (не описано в "библии"):

- плавающие константы с суффиксом F (или f) преобразуются к типу `float` и представляются четырьмя байтами вместо восьми;
- плавающие константы с суффиксом L (или l) преобразуются к типу `long double` и представляются десятью байтами вместо восьми.

Символьные константы.

Символьная константа - это некоторый символ, заключенный в одинарные кавычки. Значение символьной константы - код заключенного в кавычки символа. В Турбо Си для кодировки символов используется ASCII (American Standard Code for Information Interchange - американский стандартный код для обмена информацией).

Символы, не имеющие графического представления, а также служебные символы ' (одинарная кавычка) и \ (backslash - обратная дробная черта) представляются в соответствии с приводимой ниже таблицей (звездочками отмечены символы, не упомянутые в аналогичной таблице "библии"):

*звонок	(bell)	\a	0x07
переход на новую строку	(line feed)	\n	0x0a
возврат к началу строки	(carriage return)	\r	0x0d
горизонтальная табуляция	(horizontal tab)	\t	0x09
*вертикальная табуляция	(vertical tab)	\v	0x0b
шаг назад	(backspace)	\b	0x08
переход к новой странице печати	(formfeed)	\f	0x0c
обратная дробная черта	(backslash)	\\	0x5c
одинарная кавычка	(single quote)	\'	0x2c
*двойная кавычка	(double quote)	\"	0x22
*вопросительный знак	(question mark)	\?	0x3f
восьмеричный код символа		\DDDD	
*шестнадцатеричный код символа		\xHH	

Комбинация \DDD состоит из символа '\', за которым следуют одна, две, три или даже четыре восьмеричные цифры, задающие код требуемого символа (от 0 до 255). Точно так же используется комбинация \xHH, где HH - код, задаваемый одной или двумя шестнадцатеричными цифрами. Отметим, что как символ x, так и шестнадцатеричные цифры от a до f, могут быть набраны в любом регистре.

Константа '\0' представляет специальный символ NUL. Если за обр-
ратной дробной чертой идет символ, не входящий в число указанных, то
эта черта просто игнорируется.

В Турбо Си символ '\g' используется, в основном, лишь в форматной
строке функции `printf()`; символ '\f' используется при управлении принте-
ром.

Символьные константы в Турбо Си представляются 16-битовыми (два
байта) словами. Младший байт такого слова содержит саму константу,
старший - заполняется двоичными единицами или нулями в зависимости от
значения константы и режима работы с типом `char`:

- если по умолчанию символьный тип считается имеющим знак
(signed), то старший байт содержит знаковый (старший) разряд кода сим-
вола, то есть константы с кодами от 0 до 127 представляются как `0x00kk`,
остальные - как `0xffkk`, где `kk` - шестнадцатеричный код константы;

- если по умолчанию символьный тип знака не имеет (переключатель
`Default char type...Unsigned` в меню `Options/ Compiler/Code generation` или
опция командной строки `-k`), старший байт представления символьной кон-
станты всегда заполняется нулями.

В Турбо Си допускается также использование констант, состоящих из
двух символов, например, `'Qq'`, `'\123\xff'`. Эти константы тоже представ-
ляются 16-битовым словом, первый символ записывается в младший байт,
второй - в старший. Такие константы делают программу нетранспортабель-
ной.

Как нетрудно догадаться, введение двусимвольных констант не про-
шло даром - появилась возможность записывать константы, которые могут
быть истолкованы двояко, например, запись `'\123'` может задавать как
константу `'S'` (`0x53`), так и двусимвольную константу, состоящую из
символа с восьмеричным кодом 12 и цифры 3. В этом случае вступает в
силу упомянутое ранее соглашение о лексемах, и потому при обработке
констант, заданных с помощью восьмеричных кодов, кодом символа счита-
ется максимально длинная последовательность восьмеричных цифр. Примеры:

- константа `'\123'` состоит из одного символа `'S'`;
- константа `'\0127'` состоит из символов `'W'` и `' '` (пробел);
- константа `'\129'` состоит из символа с восьмеричным кодом 12 и
символа `'9'` (9 - не восьмеричная цифра, поэтому в код не включена);
- константа `'\2246'` вызовет сообщение об ошибке "слишком большая
числовая константа".

Строки.

Строка - это последовательность символов, окаймленная двойными
кавычками. Строки считаются значениями типа "массив символов", иници-
ированными заданными символами, и имеют статический тип. В стандарте
Си любые строки, даже полностью совпадающие, считаются различными
объектами; в Турбо Си можно установить режим, при котором совпадающие
строки отождествляются.

Компилятор завершает каждую строку нулевым байтом (специальным
символом NUL), так что конец строки можно найти просмотром ее симво-
лов. Если в строке должна находиться двойная кавычка, ей должен пред-

шествовать символ '\'. Символы, не имеющие графического представления, задаются в строке способом, описанным в предыдущем разделе. Символ '\' в конце строки текста игнорируется - это позволяет записывать строковые константы, занимающие несколько строчек исходного текста.

Турбо Си предоставляет и другой способ записи длинных строковых констант. Константа может состоять из нескольких подряд записанных элементов, каждый из которых - строка в обычном понимании. Компилятор самостоятельно склеит эти элементы в одну строку. Пример: запись строки в Турбо Си

"\nПример очень длинной строки, которую так и тянет "

"разбить на несколько кусочков, чтобы распечатка "

"программы поместилась на узкой бумаге моего принтера"

эквивалентна "стандартной":

"\nПример очень длинной строки, которую так и тянет \ разбить на несколько кусочков, чтобы распечатка программы \ поместилась на узкой бумаге моего принтера".

Этот механизм в сочетании с использованием препроцессора позволяет создавать и "составные" строковые константы, также повышающие качество текста программы, например:

```
#define PROMPT_HEAD "\nНаберите "
```

```
#define PROMPT_TAIL " , затем нажмите Return:"
```

```
static char *decimal_prompt =
```

```
PROMPT_HEAD " десятичное число" PROMPT_TAIL,
```

```
*yes_no_prompt =
```

```
PROMPT_HEAD " \"да\" или \"нет\""" PROMPT_TAIL;
```

В качестве начальных значений переменным `decimal_prompt` и `yes_no_prompt` будут присвоены адреса строк

"\nНаберите десятичное число, затем нажмите Return:" и

"\nНаберите \"да\" или \"нет\", затем нажмите Return:".

Конечно, использование описанного механизма может вызвать трудности при переносе программы в другую Си-систему.

Перечислимые константы.

Имена в списке, задающем перечислимый тип, трактуются как целые константы (см. параграфы 11.4 и 13.14).

9.4. ЗАРЕЗЕРВИРОВАННЫЕ СЛОВА

Перечисленные ниже лексемы считаются ключевыми (служебными) словами, и их не следует использовать в качестве идентификаторов:

TC asm	int	TC _ss
auto	TC interrupt	TC _AH
break	long	TC _AL
case	TC near	TC _AX
TC cdecl	TC pascal	TC _BH
char	register	TC _BL
AN const	return	TC _BX

continue		short	TC _CH
default	AN	signed	TC _CL
do		sizeof	TC _CX
double		static	TC _DH
else		struct	TC _DL
KR entry		switch	TC _DX
AN enum		typedef	TC _SI
extern		union	TC _DI
TC far		unsigned	TC _BP
float	AN	void	TC _SP
for	AN	volatile	TC _ES
KR fortran		while	TC _SS
goto		TC _cs	TC _CS
TC huge	TC	_ds	TC _DS
if	TC	_es	TC _FLAGS

Символами "AN" здесь помечены ключевые слова, не упомянутые в "библии", но входящие в ANSI C, символами "TC" - слова, специфичные для Турбо Си (имеется возможность исключить эти слова из списка - опция ANSI keywords only...On меню Options/Compiler/Source или опция командной строки -A), символами "KR" - ключевые слова, упоминаемые в библии, но отсутствующие в Турбо Си.

10. ОПИСАНИЯ

В данном разделе рассмотрены общие правила построения объявлений, область видимости имени, а также правила определения типов. Изложение достаточно формально и предназначено, в основном, для справок. Примеры и развернутые объяснения вынесены в специальные разделы: объявлениям переменных посвящен параграф 10.4; в параграфе 10.5 содержатся правила объявления и определения (это - несколько разные вещи) функций.

10.1. ИМЕНА

Имена обозначают в программе объекты, функции, типы и значения. Объявление фиксирует область видимости имени и сопоставляет с ним тип, который задает правила его использования. Объект - это область памяти, предназначенная для хранения значений. Интерпретацию значений определяет тип имени, используемого для доступа к значениям объекта. Объект, с которым объявлением сопоставлено имя, называется переменной; имя объекта при этом называется идентификатором.

10.2. НОТАЦИЯ ДЛЯ СИНТАКСИСА

В "библии по Си" использована нотация, согласно которой синтаксические понятия выделяются курсивом, а терминалы - другим шрифтом. Необязательное вхождение понятия указывается при помощи нижнего индекса "opt". По причинам, ясным любому, кто предпочитает пользоваться нашим родным "Лексиконом" или встроенным редактором Турбо Си, а не сложным редактором типа ChiWriter, при описании синтаксиса нам пришлось немного изменить нотацию. Обозначения синтаксических понятий мы будем окаймлять угловыми скобками без пробелов; пример синтаксического понятия: <инициатор>. Поэтому, скажем, сама угловая скобка, окруженная пробелами, представляет в синтаксическом правиле терминал языка, а не понятие. Для обозначения необязательности синтаксического понятия к закрывающей угловой скобке через пробел приписывается -opt. Так, запись

[<константное выражение -opt>]

означает, что в квадратных скобках, возможно, стоит константное выражение. Альтернативы синтаксических понятий обычно перечисляются в отдельных строчках текста. Кроме того, мы разделяем альтернативы еще символом '|', поскольку часто они не помещаются в одну строчку или, наоборот, занимают в строке слишком мало места.

10.3. СОСТАВ ОПИСАНИЙ

Текст программы состоит из объявлений переменных и функций, а также определений функций; все это вместе и называется описаниями:

```
<описание> ::=
    <объявление> |
    <определение функции>
```

Определение функции задает ее имя, тип возвращаемого значения и операторы, реализующие необходимые действия - так называемое тело функции. Объявления функций первоначально были изобретены для того, чтобы иметь возможность сообщить компилятору тип возвращаемого функцией значения - это необходимо при генерации кодов. В дальнейшем простое объявление функции превратилось в ее прототип; в нем сообщается дополнительно количество формальных параметров и их типы, чтобы позволить компилятору следить за правильностью обращения к функции. Объявления переменных дают компилятору информацию об именах объектов, их возможных значениях, размере занимаемой области памяти и т.п. Короче говоря, объявления диктуют компилятору интерпретацию имен, вводимых программистом.

```
<объявление> ::=
    <список атрибутов -opt>
    <список спецификаций -opt> ;
```

Обратим внимание на то, что объявление заканчивается точкой с запятой.

Каждый из списков, встречающихся в объявлении, должен быть самосогласованным - не все синтаксически допустимые комбинации компонентов описания имеют смысл. Так, опустить <список атрибутов> можно только при объявлении функции, а опустить <список спецификаций> можно только в случае, когда <список атрибутов> задаст перечисление (enum).

В списке атрибутов перечисляются компоненты объявления, общие для всех имен, перечисленных в списке спецификаций. Имена в списке спецификаций разделяются запятой, а между компонентами списка атрибутов запятая НЕ ставится:

```
<список атрибутов> ::=
    <атрибут объявления>
    <список атрибутов -opt>
<список спецификаций> ::=
    <спецификация> |
    <спецификация> , <список спецификаций> .
```

10.4. АТРИБУТЫ ОБЪЯВЛЕНИЙ

Атрибуты объявлений могут быть такими:

```
<атрибут объявления> ::=
    <атрибут памяти> |
    <атрибут типа> |
```

```
<атрибут функции> |  
cdecl | pascal |  
typedef .
```

Атрибуты памяти.

Любой идентификатор можно использовать только в части программы, называемой областью его видимости. Область видимости, а также время существования объекта, идентификатором которого является имя, задается атрибутом памяти при объявлении идентификатора.

```
<атрибут памяти> ::=  
    auto |  
    static |  
    extern |  
    register
```

Атрибут `auto` приписывается по умолчанию идентификаторам, определенным в каком-либо блоке и приводит к выделению памяти описываемому объекту. Такие объекты “живут” лишь во время работы блока - при выходе из блока занимаемая ими память освобождается, соответственно их значения будут потеряны. Атрибут `auto` не применим к идентификаторам, расположенным в файле вне функций.

Атрибут `static` сообщает компилятору о том, что нужно выделить память под объявляемый объект (когда речь идет о переменной, а не о функции) и что время жизни объекта равно времени работы программы - при выходе из блока, где находится объявление, отведенная память сохраняется, и при следующем входе в этот блок можно работать с сохраненным значением. Областью видимости такой переменной является блок, в котором она определена. Если объявление объекта находится вне функций, то атрибут `static` ограничивает область видимости имени файлом, в котором он объявлен; это правило действует и при объявлении функций.

Атрибут `register` применим лишь к переменным. Его следует понимать как атрибут `auto`, подсказывающий компилятору, что переменная используется достаточно интенсивно и потому ее желательно располагать на регистрах. Поскольку число регистров ограничено, эффективны лишь несколько первых таких объявлений. Кроме того, на регистрах могут храниться только переменные вполне определенных видов (в Турбо Си это переменные типа `int`, `char` или указатель). Для регистровых переменных имеется и другое ограничение: к ним нельзя применять адресующую операцию `&`. Если использовать атрибут разумно, то можно получить более короткую и более быструю программу.

Атрибут `extern` сообщает компилятору, что для указанного идентификатора в одном из файлов программы имеется определение, находящееся вне какого-либо блока, так что этому объекту не нужно выделять память. Если это объявление помещено внутри функции или блока, область видимости идентификатора будет функцией или блок. Если же оно находится вне функций, то его областью видимости является файл, начиная с места, где находится объявление. Считается, что если

отсутствует явный атрибут `static`, то функции описаны с атрибутом `extern`. Конечно, к функциям неприменимы атрибуты `auto` и `register`.

В объявлении можно задать не более одного атрибута памяти. Если атрибут памяти опущен, то внутри функции подразумевается атрибут `auto`, а вне - `extern`.

Областью видимости имени может быть блок, файл (в другой терминологии - модуль) и вся программа.

Блок: Переменная, объявленная в блоке, локальна для данного блока и может использоваться только в нем и в содержащихся в нем блоках. Исключениями являются метки, которые можно использовать всюду в функции, где они определены, и имена функций, область видимости которых - файл или программа. Имена формальных параметров функций считаются определенными в самом внешнем блоке функции.

Файл: Имя, определенное вне какого-либо блока, можно использовать в файле, где оно определено. Оно, тем не менее, остается неизвестным в других файлах программы (если они есть), и, если его нужно использовать там, его необходимо объявить в используемом файле с атрибутом `extern`. Если имя определено с атрибутом `static`, то оно становится "локальным для этого файла". Это значит, что всякие упоминания того же имени в других модулях будут интерпретироваться как относящиеся к другому объекту. Имена функций по умолчанию считаются определенными с атрибутом `extern`; если же описанию функции предшествует атрибут `static`, то функция локальна в файле.

Программа: Имя с атрибутом `extern` объявляет объект, определенный где-то в другом месте программы. Такой объект необходимо определить (объявить без атрибутов!) вне блоков в одном из файлов программы; определение объекта может находиться в одном файле с его объявлением. Атрибут `extern` может сопровождать определение функции.

Пример. Пусть в одном файле содержится лишь

```
char *msg = "Hello, world!";  
содержимое же другого файла:  
void main(void)  
{  
    extern char *msg;  
    printf("\n%s",msg);  
}
```

Прежде всего повторим, что атрибут `auto` неприменим в первом файле. Далее, объявление `extern char *msg` во втором файле ссылается на некоторый объект, находящийся где-то в другом месте. Область видимости идентификатора `msg` во втором файле - функция `main()`, так что если бы в этом файле были другие функции, в них этот идентификатор был бы неизвестен. Если бы объявление находилось вне функции, то идентификатор `msg` был бы известен во всех функциях, расположенных текстуально после объявления. Редактор связей во время подготовки ехе-файла подставит на место упоминания `msg` адрес объекта, определенного в первом файле.

Идентификатор может быть временно переопределен в блоке или функции. Вместо сухих и, тем не менее, малопонятных объяснений приведем пример.

```
#include <stdio.h>  
char *txt = "Приветик!";  
void main(void)
```

```

{
    printf("\n%s",txt); /* На экране: Приветик! */
    {
        char *txt = "Hello!"; /* Переопределение txt */
        printf("\n%s",txt); /* На экране: Hello! */
    }
    printf("\n%s",txt); /* На экране снова: Приветик! */
}

```

Атрибуты типа.

Имеются следующие атрибуты типа:

```

<атрибут типа> ::=
    <модификатор переменной>      |
    void                            |
    <спецификатор арифметического типа> |
    <спецификатор составного типа>    |
    enum                            |
    <typedef-имя>

```

О модификаторах переменных `const` и `volatile`, применимых, конечно, лишь при объявлении переменных, рассказывается далее.

Тип `void` отсутствует в "библии", он принадлежит современному стандарту Си. Применяется для функций и указателей.

```

<спецификатор арифметического типа> ::=
    <модификатор арифметического типа -opt>
    <спецификатор арифметического типа> |
    char | int | float | double
    <модификатор арифметического типа> ::=
    short | long | signed | unsigned

```

Арифметические типы `char`, `int`, `float` и `double` - исходные типы объектов, они служат "строительным материалом" для всех других типов. Эти типы рассматриваются еще в параграфе 11.2.

Составными типами в Си являются структуры и объединения:

```

<спецификатор составного типа> ::=
    struct | union

```

Если начало объявления имеет вид `enum <идентификатор>`, то идентификатор именуется перечислимый тип. Список констант, составляющих этот тип, можно определить где-либо в программе - до, или после, или даже внутри такого объявления.

Атрибутом типа может служить также имя, определенное где-либо в программе с помощью атрибута объявления `typedef`. Определение имени, конечно, должно располагаться в тексте до его появления в качестве атрибута типа.

Атрибут `typedef` не приводит к выделению какой-либо памяти, подобное объявление задаст так называемое `typedef-имя`, которое позже можно использовать в качестве служебного слова наравне с названиями основных или производных типов. Областью видимости такого имени является файл.

```

<typedef-имя> ::= <идентификатор>

```

Например, после описаний

```
typedef int MILES, *KlickSP;
typedef struct {double re,im;} complex;
```

конструкции

```
MILES distance;
extern KlickSP metricp;
complex z, *zp;
```

становятся вполне допустимыми; тип переменной `distance` - `int`, тип переменной `metricp` - указатель на `int`, `z` - определенная выше структура, а `zp` - указатель на такую структуру.

Атрибут `typedef` не вводит новые типы, а вводит лишь синонимы для типов, которые можно было бы задать и другим способом. Так, в приведенном выше примере тип переменной `distance` совершенно такой же, как у любого другого объекта типа `int`.

Атрибуты функций.

Фирма Borland изобрела для Турбо Си три атрибута функций (они отсутствуют как в "библии", так и в ANSI Си):

<атрибут функции> ::=

```
interrupt | near | far .
```

Атрибут `interrupt` применяется в тех случаях, когда предполагается написать собственный обработчик прерываний. Этот атрибут предписывает компилятору генерировать специальные коды для функции: при входе в нее спасаются все регистры процессора, выход из нее - специальная команда `IRET`. Мы планируем включить в нашу серию выпуск на эту тему, так что здесь скажем лишь, что перед компиляцией такой функции нужно отключить как предупреждение о стеке (`Stack warning`), так и использование регистровых переменных.

Атрибуты `near` и `far` также используются для специальных целей: они предписывают компилятору генерировать разные коды для входа в функцию и выхода из нее. Настоятельно рекомендуем не пользоваться этими атрибутами до тех пор, пока вы не разберетесь в том, как и какие коды генерируются. Воспользуйтесь опцией `-S` командной строки компилятора `tcc`, чтобы получить представление программы на языке ассемблера.

Атрибуты `cdecl` и `pascal`.

Атрибуты `cdecl` и `pascal` - также принадлежность Турбо Си: они не входят в ANSI Си и, тем более, не упоминаются в "библии". В объявлении может стоять лишь один из них.

Атрибут `cdecl`, примененный к переменной, сообщает компилятору, что нужно использовать принятые в Си правила интерпретации ее идентификатора: чувствительность к регистру - большие и маленькие буквы считаются различными, и генерация символа подчеркивания `'_'` (для редактора связей), т.е. скажем, из объявления

```
int cdecl PaScAL;
```

будет построено имя `_PaScAL`.

Напротив, атрибут `pascal` говорит, что нужно использовать правила Паскаля: имя представляется большими буквами и без подчеркика, т.е. скажем, из объявления

```
int pascal PaScAl;
```

будет построено имя `PASCAL`.

Когда один из этих атрибутов предшествует определению функции, он управляет дополнительно способом передачи фактических параметров:

- атрибут `pascal` предписывает соблюдать соглашения, принятые в Паскале, т.е. первым в стек заносится первый параметр, вторым - второй и т.д., чистит стек вызванная функция;

- при атрибуте `cdecl` первым в стек будет занесен последний параметр и стек чистит вызывающая функция.

Атрибут `pascal` был придуман, по-видимому, для возможного сопряжения Турбо Си с Турбо Паскалем. Однако неизвестны средства сколь-нибудь безболезненной реализации такого сопряжения. Поэтому мы настоятельно рекомендуем (хотя бы на первых порах) не пользоваться этими атрибутами. В меню имеется возможность установить атрибут `pascal` для функций по умолчанию - не делайте этого никогда, не оберетесь хлопот!

Атрибут `typedef`.

Как уже говорилось выше, атрибут `typedef` используется для введения синонимов типов. Это позволяет "спрятать" от пользователя конкретное представление типов данных. Кроме того, `typedef` часто используется для облегчения понимания объявлений.

10.5. СПЕЦИФИКАЦИИ В ОБЪЯВЛЕНИИ

Спецификация в объявлении переменной может содержать выражение-инициатор для этой переменной:

```
<спецификация> ::=
```

```
<спецификация имени> <инициатор -opt> .
```

Инициаторы рассматриваются в следующем параграфе.

Вид спецификации имени определяет, что именно объявляется: просто переменная, функция или указатель и т. д.

```
<спецификация имени> ::=
```

```
<идентификатор> |
(<спецификация>) |
<спецификация указателя> |
<спецификация массива> |
<спецификация составного типа> |
<спецификация перечислимого типа> |
<спецификация функции>
```

Если спецификация - это просто <идентификатор>, то объявляется переменная, тип которой задается атрибутами объявления.

Спецификация в скобках применяется при объявлении чего-нибудь сложного, часто - указателя на функцию. Спецификация указателя выглядит так:

```

<спецификация указателя> ::=
    <модификатор указателя -opt>
        * <модификатор переменной -opt>
            <идентификатор> .

```

Модификаторы указателя - принадлежность Турбо Си, они используются для "тонкой настройки" программы на хитрые приложения.

```

<модификатор указателя> ::=
    near | far | huge |
    _es | _ss | _cs | _ds

```

В Си указатели и массивы - "близнецы-братья": идентификатор массива является указателем на некоторую область памяти, причем тип этого указателя задается типом массива. Кроме того, все выражения, в которых фигурируют элементы массива, всегда можно заменить на коды, работающие с указателем; это, в частности, означает, что в программах вполне можно было бы обходиться без массивов. Единственное ощутимое различие между указателем и массивом состоит в том, что под объявленный массив компилятор сам отводит необходимую память, а в "безмассивной" программе о памяти пришлось бы заботиться нам.

```

<спецификация массива> ::=
    <модификатор указателя -opt>
        <идентификатор>
            [ <константное выражение> ]

```

Составными типами в Си являются структуры и объединения. Что именно объявляется, задается атрибутом объявления: если нужно объявить структуру, атрибутом является `struct`, чтобы объявить объединение, используется атрибут `union`. Спецификация составного типа задает его поля; идентификатор, если он есть, нужен для объявления переменных. Поля составного типа могут быть произвольными переменными, в частности, снова структурой или объединением; поле не может быть функцией, хотя указателем на функцию ему быть разрешается, так что особых ограничений здесь нет. Кроме того, можно объявить составной тип с битовыми полями - именно поэтому при описании синтаксиса возникают варианты:

```

<спецификация составного типа> ::=
    <идентификатор -opt> { <список полей> }
<список полей> ::=
    <спецификация поля> ;
    <список полей -opt>
<спецификация поля> ::=
    <атрибут типа> <спецификация> |
    <спецификатор арифметического типа -opt>
    <идентификатор -opt> :
    <константное выражение> .

```

Перечислимые типы - достаточно недавнее приобретение Си. Они являются хилым вариантом скалярных типов из языков типа Паскаль и, поскольку ни во время компиляции, ни, тем более, во время работы программы никакие сколько-нибудь разумные проверки не производятся, их используют, фактически, лишь для повышения "читаемости" программы.

```

<спецификация перечислимого типа> ::=
    <список значений -opt>

```



```

<список значений> ::=
    { <список имен с инициаторами> }
<список имен с инициаторами> ::=
    <имя с инициатором> |
    <имя с инициатором> ,
    <список имен с инициаторами>
<имя с инициатором> ::=
    <идентификатор> |
    <идентификатор> = <целая константа>

```

Именам, перечисленным в списке значений, присваиваются целые значения, сами имена компилятор трактует как целые константы.

Функции - слишком важная материя, чтобы здесь так уж вскользь и по-быстрому рассказать о них, поэтому объявлениям и определениям функций посвящена специальная глава. Здесь мы приведем для полноты только синтаксис, ограничившись лишь предварительными комментариями.

```

<спецификация функции> ::=
    <декларация функции> |
    <прототип функции>

```

Декларация функции (вместе с атрибутами объявления, конечно) говорит компилятору только о типе возвращаемого функцией значения:

```

<декларация функции> ::=
    <идентификатор> ()

```

Прототип (перед которым также стоят атрибуты объявления) дополнительно говорит о количестве аргументов и их типах:

```

<прототип функции> ::=
    <идентификатор> ( void ) |
    <идентификатор> ( <список аргументов> )

```

Список аргументов может состоять лишь из типов, перечисленных через запятую. Можно также сопровождать каждый из типов или некоторые из них именем аргумента - если имена выбраны разумно, это иногда позволяет догадаться о смысле аргумента лишь по прототипу функции.

```

<список аргументов> ::=
    ...
    <тип аргумента> |
    <тип аргумента> , <список аргументов> |
    <объявление> |
    <объявление> , <список аргументов>

```

Конечно, здесь допустимы не все компоненты объявления.

```

<тип аргумента> ::=
    <список атрибутов типа>
<список атрибутов типа> ::=
    <атрибут типа>
    <список атрибутов типа -оп>

```

Мы настоятельно рекомендуем полное прототипирование ВСЕХ ваших функций. Не ленитесь также включать (директивой `#include` препроцессора) системные `header`-файлы, чтобы компилятор получил прототипы всех используемых библиотечных функций. Это убережет вас от множества ошибок, поскольку на них вам укажет компилятор!

10.6. ИНИЦИАЛИЗАЦИЯ

Любой спецификатор может задать начальное значение объявляемого идентификатора. Инициатор начинается с символа '=' и состоит из выражения или списка значений, заключенного в фигурные скобки. Обратите внимание на то, что список инициаторов в фигурных скобках может заканчиваться запятой (так в "библии").

```
<инициатор> ::=
    = <выражение> |
    = { <список инициаторов> } |
    = { <список инициаторов> , }
<список инициаторов> ::=
    <инициатор> |
    <инициатор> ,
    <список инициаторов> |
    { <список инициаторов> }
```

Все выражения инициаторов для статических или внешних переменных должны быть константными выражениями или выражениями, сводящимися к ссылкам на предварительно описанные переменные, при этом допускается смещение от них на величину, опять же заданную константным выражением. Автоматические и регистровые переменные могут инициализироваться произвольными выражениями, включающими константы и предварительно описанные переменные и функции.

Если статическая или внешняя переменная (объявленная вне функций) не инициализирована, то гарантируется, что она получит значение 0; неинициализированные же автоматические или регистровые переменные будут содержать непредсказуемый "мусор".

Если инициатор относится к скалярному типу (т.е. указателю или любому объекту арифметического типа), то он представляет собой единственное выражение, возможно, в скобках. Начальное значение такого объекта определяется этим выражением, при этом выполняются те же преобразования, что и при присваивании.

Если инициатор относится к составному объекту (структуре или массиву), то он состоит из списка выражений, отделенных друг от друга запятыми, заключенного в фигурные скобки; эти выражения - инициаторы для элементов, записанные в порядке перечисления элементов или увеличения индексов. Если составной объект содержит составные подобъекты, то это правило распространяется рекурсивно и на них. Если в списке инициаторов меньше, чем элементов в объекте, то оставшиеся принимают нулевые значения. В "библии" автоматические составные объекты и объединения инициализировать не разрешается, в Турбо Си это делать можно.

Скобки можно толковать следующим образом. Если инициатор начинается с левой скобки, то последующий список инициаторов, разделенных запятыми, задает значения элементов составного объекта; если инициаторов больше, чем элементов, - это ошибка. Если инициатор не начинается со скобки, то из списка берется столько элементов для очередного объекта, сколько элементов в объекте; оставшиеся элементы списка используются для инициализации следующих элементов объекта, в состав которого данный входит как часть.

Для инициализации элементов символьного массива можно использовать строку. Это некоторая сокращенная форма инициализации, где последовательность символов строки иницирует элементы массива.

Пример. Описание

```
int x[] = { 1, 3, 5};
```

объявляет и инициализирует `x` как одномерный массив из трех элементов, так как размер его не задан, но есть три инициатора.

```
float y[4][3] = {
    { 1, 3, 5},
    {2, 4, 6 },
    {3, 5, 7 },
};
```

В такой инициализации скобки расставлены полностью: 1, 3, 5 иницируют первую строку массива `y[0]`, т.е. `y[0][0]`, `y[0][1]` и `y[0][2]`. Аналогично, две следующие строки иницируют `y[1]` и `y[2]`. Инициализация заканчивается ранее, чем требуется, поэтому `y[3]` иницируется нулями. Точно того же эффекта можно достичь описанием

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

С фигурной скобки начинается инициатор для `y`, а не для `y[0]`, поэтому из списка используются три элемента. Аналогично следующие три элемента из списка берутся для `y[1]`, а затем для `y[2]`. Описание же

```
float y[4][3] = {
    { 1 }, {2}, {3}, {4}
};
```

инициализирует первый столбец `y` (считая его двумерным массивом), а оставшиеся столбцы заполняются нулями.

Наконец, описание

```
char msg[] = "Syntax error on line %s\n";
```

задает символьный массив, элементы которого инициализированы символами строки.

10.7. СМЫСЛ ОБЪЯВЛЕНИЙ

Каждое объявление предполагает, что если конструкция такого вида, как это объявление, появляется в выражении, то она соответствует объекту указанных типа и атрибута памяти. Любое объявление содержит точно один идентификатор - именно этот идентификатор и объявляется.

Если в объявлении находится просто идентификатор, то его тип задается атрибутами объявления.

Спецификация в круглых скобках подобна спецификации без скобок, однако скобки могут изменять смысл сложных объявлений.

Представим себе объявление вида `T D1`, где `T` - атрибут типа (например, `int`), а `D1` - спецификация. Это объявление определяет, что идентификатор имеет тип `T`, если `D1` - просто идентификатор. Например, тип `x` в `int x` есть просто `int`. Если спецификация `D1` имеет вид `*D`, то тип идентификатора - ссылка на объект типа `T`. Если `D1` имеет вид `const *D`, то тип

содержащегося в D значения - неизменяемая ссылка на объект типа T. Если же D1 имеет вид *const D, то D - ссылка на неизменяемый объект типа T.

Если D1 имеет вид D(), то упомянутый идентификатор имеет тип "функция, возвращающая T".

Если D1 имеет вид D[константное выражение] или D[], то упомянутый идентификатор имеет тип "массив элементов типа T". Эпитет "константное" в первом варианте объявления означает, что выражение может быть вычислено во время компиляции; такое выражение обязано иметь тип int. Если рядом стоят несколько спецификаций "массив", то порождается многомерный массив; константное выражение, определяющее границы массива, можно опускать только для первого элемента этой последовательности. Такой пропуск осмыслен, если это внешний массив и фактическое его объявление, резервирующее память, приводится где-то в другом месте программы. Первое константное выражение можно опускать и в том случае, если за объявлением следует инициализация. В этом случае размер определяется количеством заданных начальных значений.

Массивы можно строить на основе любого из основных типов, указателей, записей или объединений, других массивов (порождая многомерные массивы).

Однако не все возможности, допускаемые приведенным выше синтаксисом, на самом деле можно использовать. Имеются такие ограничения: функции не могут возвращать массивы или функции, хотя, конечно, они могут выдавать ссылки на эти объекты; не существует массивов функций, хотя могут быть массивы ссылок на функции. Аналогично, записи или объединения не могут содержать функции, но могут содержать ссылки на функции.

Например, объявление

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

описывает целое i, указатель ip на целое, функцию f, возвращающую целое, функцию fip, возвращающую ссылку на целое, и указатель pfi, содержащий ссылку на функцию, возвращающую целое. Особенно полезно сравнить два последних случая. Толкование *fip() таково: *(fip()), так что объявление предполагает (а в некоторых конструкциях в выражении и требует) обращение к функции fip, а затем косвенное обращение через результат (ссылку) для получения целого. В объявлении (*pfi) () дополнительные скобки необходимы, это требуется для того, чтобы через указатель на функцию задать обращение за функцией, с которой затем идет обращение; последнее обращение даст в результате целое значение.

Еще один пример:

```
float f[17], *fp[17];
```

Здесь описаны массив чисел типа float и массив указателей на такие числа. Наконец, в объявлении

```
static int x3d[3][5][7];
```

вводится статический трехмерный массив целых чисел размером 3*5*7. Более детально: x3d - это массив из трех элементов, каждый элемент которого - массив из пяти массивов, а в каждом таком массиве уже по семь целых чисел. В выражении может появиться и иметь смысл любая из таких

конструкций: `x3d`, `x3d [i]`, `x3d[i][j]`, `x3d[i][j][k]`. Первые три из них имеют тип "массив", а последняя - `int`.

10.8. НАЗВАНИЯ ТИПОВ

В программе в двух местах желательно употребление "названий" для типов данных: для задания явного преобразования типа и в качестве аргумента для "функции" `sizeof`. Это можно сделать с помощью "имени типа", такое имя вводится с помощью объявления некоторого фиктивного безымянного объекта нужного типа, причем имя объекта вообще отсутствует.

```
<имя типа> ::=
    <атрибут типа> <абстрактный описатель>
<абстрактный описатель> ::=
    <пусто> |
    ( <абстрактный описатель> ) |
    * <абстрактный описатель> ( ) |
    <абстрактный описатель>
    [ <константное выражение -opt> ]
```

Чтобы избежать двусмысленности, в конструкции

```
( <абстрактный описатель> )
```

абстрактный описатель не должен быть пустым. При таком ограничении возможно единственным образом локализовать те места в абстрактном описателе, где должен был бы появиться идентификатор, если эта конструкция встретится в каком-либо объявлении. Поименованный тип - это тип, в объявление которого входит идентификатор. Например, объявления

```
int
int *
int *[3]
int (*)[3]
int*()
int(*)()
```

задают соответственно типы: целое, указатель на целое, массив из трех указателей на целое, указатель на массив из трех целых, функция, возвращающая указатель на целое, указатель на функцию, возвращающую целое.

10.9. ОПРЕДЕЛЕНИЕ ФУНКЦИИ

Определение функции состоит из ее заголовка и тела:

```
<определение функции> ::=
    <атрибут памяти> <заголовок функции>
    <список объявлений аргументов -opt>
    { <тело функции> } .
```

Из атрибутов памяти здесь допустимы лишь `extern` и `static`, задающие область видимости для имени функции.

В заголовке мы обязаны указать тип возвращаемого функцией значения, ее имя, количество и имена параметров. Информацию о типах фор-

мальных параметров либо дает <список объявлений аргументов>, либо она включается в заголовок - и тогда <список объявлений аргументов> отсутствует.

```

<заголовок функции> ::=
    <идентификатор> ( void )           |
    <идентификатор>
      ( <список имен аргументов -opt> ) |
    <идентификатор>
      ( <список формальных параметров> )
<список имен аргументов> ::=
    ...                               |
    <имя>                             |
    <имя> , <список имен аргументов>
<список объявлений аргументов> ::=
    <объявление> ;
    <список объявлений аргументов -opt>
<список формальных параметров> ::=
    ...                               |
    <объявление>                     |
    <объявление> ,
    <список формальных параметров>

```

Представленные здесь два способа определения функции соответствуют старому и новому "стилю" в Си: включение типов аргументов прямо в заголовок функции - новый стиль, объединяющий определение функции с ее прототипированием. К функции, определенной таким образом, можно без опасений обращаться (но только в том же файле и, конечно, текстуально после определения) без прототипа - компилятор использует в качестве прототипа ее заголовок.

11. ОПИСАНИЯ ПЕРЕМЕННЫХ

В данной главе мы иллюстрируем правила объявления переменных, используемых в программе, а также повторяем из предыдущего раздела информацию, относящуюся к переменным.

11.1. ОБЪЕКТЫ И ИХ ИМЕНА

Объекты - это данные, с которыми имеет дело программа. Объекты располагаются в некоторой области памяти, размер которой определяется типом объекта, ее конкретное содержимое называется значением объекта. Тип объекта задает совокупность его допустимых значений и правила их интерпретации. Адрес области памяти, где располагается объект, является одной из характеристик объекта, причем адрес каждого объекта уникален и позволяет получить его значение.

Важным примером является объект, значениями которого являются адреса объектов определенного типа. Его значения называются также указателями, про объект с такими значениями говорят, что он имеет ссылочный тип. В памяти IBM PC указатели представляются двумя или четырьмя байтами.

Объекты могут быть созданы как во время работы программы, так и на этапе компиляции. Объекты, создаваемые на этапе компиляции, должны быть объявлены - только через объявление компилятор может узнать о существовании объекта, о необходимой для него памяти и пр. Объявление, кроме того, сопоставляет с объектом имя; пара "объект+имя" называется переменной, имя в этом случае называется идентификатором переменной. Объявление связывает с идентификатором два его атрибута: класс памяти и тип именуемого им объекта. В каждый момент выполнения программы каждое имя может быть связано только с одним объектом. Время существования этой связи определяется атрибутом памяти переменной. Атрибут памяти определяет также местонахождение в памяти ЭВМ соответствующего объекта.

Выражение задает способ получения нового значения, называемого результатом (вычисления) выражения из известных значений, называемых операндами. Способ получения результата описывается с помощью фиксированного в языке набора операций. Результат выражения может быть сделан новым содержимым какого-либо объекта при помощи операции присваивания.

Практически во всех современных языках программирования, и Си не является исключением, идентификатор переменной интерпретируется по-разному в зависимости от места, в котором оно встретилось в программе. В выражениях идентификатор переменной обозначает соответствующее значение, в операции же присваивания слева от знака равенства он адресует сам объект.

Синтаксический способ задания адреса объекта будем называть ссылочным выражением, а значение такого выражения - ссылкой (в англоязычной литературе по Си и в сообщениях компилятора Турбо Си используется термин *Lvalue*). Простейшим примером ссылочного выражения служит идентификатор переменной, стоящий слева в операции присваивания. Однако, идентификатор переменной является лишь очень частным случаем. Любое выражение ссылочного типа (см. ниже) можно поместить слева от знака равенства в операции присваивания. Поэтому в программе могут встретиться достаточно странные на первый взгляд конструкции, например, такая:

$f(a) \rightarrow \text{next} = h(b);$,

здесь $f()$ - функция, возвращающая ссылку на структуру с полем *next*, которому присваивается значение, возвращаемое функцией $h()$.

11.2. ОБЪЕКТЫ ИСХОДНЫХ ТИПОВ

В Си предусмотрены четыре исходных типа объектов: **char** (символьные), **int** (целые), **float** (их часто называют одинарными действительными) и **double** (двойные действительные или действительные с двойной точностью). Объектам символьного типа (**char**) отводится достаточно памяти, чтобы хранить любой из элементов множества символов ASCII (во всех современных компиляторах - один байт). Если некоторый символ из этого множества хранится в символьной переменной, то ее значение эквивалентно целому числу, представляющему данный символ.

Объекты этих основных типов часто интерпретируются как числа, поэтому эти типы называют арифметическими. Типы **char** и **int** (вместе с модификаторами) называют при этом целыми типами, а типы **float** и **double** - плавающими.

В Турбо Си объявление любого объекта арифметического типа можно дополнить одним из следующих модификаторов:

Модификатор	Его действие
-------------	--------------

short	уменьшает область допустимых значений
long	увеличивает область допустимых значений

Кроме того, объявление объекта целого типа можно уточнить еще одним из следующих модификаторов:

Модификатор	Его действие
-------------	--------------

signed	старший бит используется как знаковый
unsigned	старший бит не используется как знаковый

Если в объявлении тип опущен (в наличии только один или несколько модификаторов), подразумевается **int**.

Приведем таблицу, в которой перечислены комбинации основных типов данных и их модификаторов:

Тип	Размер	Область значений в байтах	
char	1	от -127 до 127	signed
char	1	от -127 до 127	unsigned
char	1	от 0 до 255	

int	2	от -32768 до 32767	signed
int	2	от -32768 до 32767	unsigned
int	2	от 0 до 65535	
short int	2	от -32768 до 32767	unsigned
short int	2	от 0 до 65535	
long int	4	от -2147483648 до 2147483647	signed
long int	4	от -2147483648 до 2147483647	unsigned
long int	4	от 0 до 4294967295	
float	4	от 3.4E-38 до 3.4E+38	
		и от -3.4E+3 до -3.4E-3	
long float	8	от 1.7E-30 до 1.7E+30	
		и от -1.7E-30 до -1.7E+30	
double	8	от 1.7E-30 до 1.7E+30	
		и от -1.7E-30 до -1.7E+30	
long double	10	от 3.4E-493 до 3.4E+493	
		и от -3.4E-493 до -3.4E-493	

Примечания:

1. Трактовка типа `char` без модификатора управляется переключателем `Default char type` в меню `Options/ Compiler/ Code generation`. Если переключатель установлен в `Signed` (стандартное значение; этот факт отражен в таблице), то описатель `char` означает `signed char`. Если же переключатель установлен в `Unsigned` (или задана опция командной строки `-K`), по умолчанию предполагается наличие модификатора `unsigned`.

2. Тип `long double` в Турбо Си версии 1.5 трактуется точно так же, как `double`.

Из исходных типов можно построить концептуально бесконечное множество производных типов. Для этого имеются следующие средства:

массивы - их можно построить для большинства типов;

функции, возвращающие объект заданного типа;

указатели на объекты заданного типа;

объединения, способные содержать объекты одного из нескольких заданных типов.

Как правило, методы построения составных типов можно применять рекурсивно.

11.3. ССЫЛОЧНЫЕ ТИПЫ И МАССИВЫ

Объекты, предназначенные для адресации объектов, необходимы в каждой программе, использующей сколько-нибудь сложные структуры данных. В Си для этой цели используются так называемые указатели (`pointer`), переменные особого - ссылочного - типа, в которых хранятся адреса памяти. С каждым указателем сопоставляется тип тех данных, адреса которых в нем могут находиться. Это делает возможной так называемую адресную арифметику: когда в программе мы увеличиваем указатель на `n`, его значение увеличивается фактически на `n*(размер типа)`, где размер типа - количество байтов, занимаемых в памяти одним объектом. Общий синтаксис описания переменной ссылочного типа:

<спецификация указателя> ::=

<модификатор указателя -opt>
 *<модификатор переменной -opt>
 <идентификатор>

<модификатор указателя> ::=
 near | far | huge |
 _es | _ss | _cs | _ds

Вместо имени типа здесь может стоять спецификатор **void**; модификаторы указателя не входят в ANSI Си и, конечно, не упоминаются в "библии".

Приведем несколько примеров.

```
int *iptr; /* указатель на переменную целого типа */
char *cptr; /* указатель на символьную переменную */
struct complex *pcmp; /* указатель на
                        /* структуру complex, */
                        /* описанную где-то */
                        /* в другом месте */
struct scr { /* Объявление структуры scr */
  int row,col; /* с полями row и col; */
} cell /* объявление переменной cell */
      = {11,22}; /* и ее инициализация. */
struct scr *pscr /* Объявление указателя */
              /* на структуру scr */
              = &cell; /* и его инициализация адресом */
                    /* переменной cell */
```

При инициализации указателя символ '&' вовсе не обязателен:
 char *msg = "Здесь был Федя!";

Компилятор выделит область памяти, разместит в ней заданную строку и сделает значением указателя msg адрес начала выделенной области.

Для ссылочных типов в Турбо Си введены три модификатора **near**, **far** и **huge**, выбираемые с учетом используемой модели памяти. Вот как зависит размер ссылочной переменной от модификатора:

Модификатор Размер (в байтах)

near	2
far	4
huge	4

Модификаторы **huge** и **far** определяют, что ссылка представляется четырехбайтным указателем. Однако в отличие от **far** значения **huge**-указателя нормализованы, и потому арифметические операции с **huge**-указателем изменяют как его сегментную часть, так и смещение; у **far**-указателей изменяется лишь смещение. Удобства, предоставляемые нормализацией, настолько значительны, что иногда заставляют смириться с потерей времени на требуемые для нее преобразования:

- операции сравнения (**=**, **!=**, **<**, **<=**, **>**, **>=**) всегда дают естественно предсказуемый результат для **huge**-указателей, это не выполнено для **far**-указателей; - значения **huge**-указателя могут адресовать всю память (1 Мб), а значения **far**-указателя - лишь 64 Кб.

В качестве типа указателя может стоять **void**, что значит указатель неопределенного типа. Такой указатель не может участвовать в операциях, где требуется знание размера адресуемого объекта. Зато ему можно присво-

ить значение указателя любого типа и указателю любого типа можно присвоить его значение.

Кроме того, после атрибутов, определяющих тип указателя, может стоять еще один из следующих "эпитетов": `_es`, `_ss`, `_cs` или `_ds`. Такие модификаторы используются при определении специальных версий `near`-указателей для разных типов данных. Подобные указатели представляют собой 16-битовые значения, для получения адреса они ассоциируются с регистром `es`, `ss`, `cs` или `ds`, указанным при объявлении.

Примеры:

```
char _cs *s;      /* в кодовом сегменте */
int _ss *ix;     /* в сегменте стека */
long _ds l[4];   /* в сегменте данных */
char _cs m[8];   /* в es-сегменте */
```

Мы здесь не будем рассказывать о работе с такими указателями, поскольку те, кто способен ими воспользоваться, уже не нуждаются в нашем руководстве.

Массивы в Си определяются тем, что за именем переменной следует количество элементов каждой размерности в квадратных скобках. Нижняя граница каждой размерности считается равной нулю. Это связано с тем, как компилируются выражения типа `a[i]`. Общий синтаксис объявления массива:

```
<спецификация массива> ::=
    <модификатор указателя -opt>
    <идентификатор>
    [ <константное выражение> ]
```

Каждая размерность массива представляется отдельной парой квадратных скобок. Массивы, конечно, тоже можно инициализировать. Приведем несколько примеров:

```
int numbers[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
char dec_digits[10] = {'0', '1', '2', '3', '4',
                      '5', '6', '7', '8', '9'};
struct complex zz[2] /* Массив из двух элементов, */
                   /* каждый из которых - структура */
                   = { /* Инициализация массива: */
                      {1.0, 2.0}, /* значение zz[0] */
                      {3.4} /* значение zz[1] */
                    };
/* Компилятор поймет по описанию структуры complex, что
   числа 3 и 4 нужно преобразовать в double */
double t[2][3] = {
    {1,2,3}, /* значения t[0][0..2] */
    {4,5,6} /* значения t[1][0..2] */
};
```

Последний пример заодно иллюстрирует расположение в памяти многомерных массивов: самый правый индекс меняется быстрее всего.

Большим удобством Си является то, что при инициализации массивов не всегда обязательно указывать истинные размерности: компилятор сам подсчитает их. Перепишем приведенные примеры, используя это соглашение:

```
int numbers[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
char dec_digits[] = {'0', '1', '2', '3', '4',
```

```

        '5', '6', '7', '8', '9');
struct complex zz[ ] = {
    {1.0, 2.0}, /* значение zz[0] */
    {3.4} /* значение zz[1] */
};
double t[][ ] = {
    {1,2,3}, /* значения t[0][0..2] */
    {4,5,6} /* значения t[1][0..2] */
};

```

Конечно, если инициализируется не весь массив, то необходимо про-
ставить требуемые размерности. При этом неинициализированная часть
будет заполнена нулями.

Строки в Си считаются массивами символов, конец такого массива
помечается элементом, содержащим ноль. Это соглашение часто служит
источником ошибки при обращении к библиотечным функциям, работаю-
щим со строками: программисты забывают отвести лишний элемент под
символ '\0'.

11.4. ПЕРЕЧИСЛИМЫЕ ТИПЫ

Перечисления позволяют программисту определить список имен, с
каждым из которых сопоставлено одно какое-либо (обязательно целое!)
значение. Элементы списка не могут появиться в другом списке. Общий
синтаксис объявления перечисления таков:

```

<спецификация перечислимого типа> ::=
    <список значений -opt>
<список значений> ::=
    { <список имен с инициаторами> }
<список имен с инициаторами> ::=
    <имя с инициатором> |
    <имя с инициатором> ,
    <список имен с инициаторами>
<имя с инициатором> ::=
    <идентификатор> |
    <идентификатор> = <целая константа> .

```

Приведем несколько примеров:

```

enum boolean {FALSE, TRUE};
enum logic {false, true, both, none};
enum week { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

```

Обратим внимание на то, что "ложь" и "истина" в типах **boolean** и
logic обозначены по-разному (напомним, что в Си прописные и строчные
буквы различаются). Если бы, например, значения типа **boolean** были
заданы строчными буквами, то **boolean** и **logic** нельзя было бы использовать
в одной программе.

По умолчанию первый элемент списка кодируется значением 0, вто-
рой - значением 1 и т.д. Программист может управлять присваиваемыми
значениями либо явно указывая их для каждого элемента списка, либо
указав требуемое значение для какого-то одного элемента - для других они
будут автоматически увеличиваться на единицу. Пусть, например, мы хотим
считать воскресенье не нулевым днем недели, а первым, понедельник -

вторым и т.д. Тогда в приведенном примере нам потребуется явно задать значение для элемента `Sun`:

```
enum week { Sun=1, Mon, Tue, Wed, Thu, Fri, Sat };
```

Заодно мы сказали, что понедельник - это второй день недели, вторник - третий и т.д. Если нам почему-либо не нравится "буржуазная" нумерация дней недели, достаточно написать:

```
enum week { Sun=7, Mon=1, Tue, Wed, Thu, Fri, Sat };
```

Конечно, можно изменить и порядок идентификаторов в списке:

```
enum week { Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun };
```

В планируемом выпуске о работе с клавиатурой приведен очень большой список идентификаторов, в котором для каждого элемента списка явно сказано, какое именно значение с ним нужно сопоставить. Заметим уже сейчас, что эти значения не обязательно все различны.

После того, как определен перечислимый тип, мы можем объявить переменную (или функцию) этого типа. Примеры:

```
enum week days; /* Переменная days того из типов week, */
                /* который мы оставили в программе */
enum logic rule; /* Предполагается, что переменная rule */
                /* принимает одно из логических значений */
```

Когда эти переменные будут участвовать в выражениях, они преобразуются к целому типу. Так, если значением переменной `rule` является `none`, то в выражении будет фигурировать число 3.

В соответствии с приведенным синтаксисом мы можем ввести безымянный тип, ограничившись объявлением соответствующей переменной:

```
enum { draw, defeat, victory } result, game_value;
```

однако в этом случае придется ограничиться лишь одним объявлением, т.к. повторить его компилятор не позволит - ему не понравится переопределение одного и того же имени. Так, строки

```
enum { draw, defeat, victory } result;
enum { draw, defeat, victory } game_value;
```

компилятор отвергнет, выдав сообщение "**Redeclaration of '<имя>'**" о каждом имени из списка.

Можно, конечно, и совместить определение типа с объявлением переменной. Следующая пара строк не вызовет возражений компилятора:

```
enum game_result { draw, defeat, victory } result,
enum game_result game_value;
```

первая строка вводит перечислимый тип с именем `game_result` и объявляет переменную `result` этого типа, а вторая объявляет переменную с именем `game_value` того же типа.

Наконец, можно объявить переменную перечислимого типа до какого-либо определения самого типа:

```
enum ret_code ret_value;
```

```
.....
```

```
enum ret_code { OK, BAD_PARAMS, OUT_OF_MEMORY };
```

11.5. СТРУКТУРЫ И ОБЪЕДИНЕНИЯ

Структуры позволяют программисту держать вместе семантически связанные данные. Эти данные называются полями, каждое поле является основным или пользовательским типом. Таким образом, полем структуры может быть массив, перечислимый тип, другие структуры, объединения. Для определения структур в Си используется следующий синтаксис:

```
struct <имя структуры> {
    <тип 1> <имя поля 1>;
    <тип 2> <имя поля 2>;
    .....
    <тип n> <имя поля n>;
};
```

Имена полей структур могут совпадать с обычными переменными. Однако имена элементов должны отличаться от имени типа структуры, в которую они входят.

Приведем несколько примеров:

```
struct complex {
    double real, imagin;
};
struct pixel {
    int col_coord, row_coord;
    enum COLORS color;
};
```

Теперь можно определить переменные, именующие объекты описанных типов:

```
struct complex z,*pz; /* z - структура complex, */
                      /* pz - указатель на */
                      /* такую структуру */
struct pixel screen[80*25]; /* Массив структур */
```

Любителям экономии предназначены структуры с битовыми полями. Общий синтаксис их описания:

```
struct <имя структуры> {
    <тип 1> <имя поля 1> : <количество битов в поле 1>;
    <тип 2> <имя поля 2> : <количество битов в поле 2>;
    .....
    <тип n> <имя поля n> : <количество битов в поле n>;
};
```

поля такой структуры располагаются в памяти справа налево (от младших битов к старшим). Приведем несколько примеров:

```
struct two_bytes {
    unsigned int byte1 : 8; /* младший байт */
    unsigned int byte2 : 8; /* старший байт */
};
struct ftme { /* структура из Ю.Н. библио- */
              /* течного файла Турбо Си */
    unsigned ft_tsec : 5; /* Two second interval*/
    unsigned ft_min : 6; /* Minutes */
};
```

```

unsigned ft_hour : 5; /* Hours */
unsigned ft_day : 5; /* Days */
unsigned ft_month : 4; /* Months */
unsigned ft_year : 7; /* Year */
};
struct kstatus { /* Структура, представляющая */
/* состояние клавиатуры */
unsigned right_shift : 1; /* Нажат правый Shift. */
unsigned left_shift : 1; /* Нажат левый Shift. */
unsigned ctrl_shift : 1; /* Нажат Ctrl. */
unsigned alt_shift : 1; /* Нажат Alt. */
unsigned scroll_state : 1; /* Включен Scroll Lock. */
unsigned num_state : 1; /* Включен Num Lock. */
unsigned caps_state : 1; /* Включен Caps Lock. */
unsigned ins_state : 1; /* Включено состояние */
/* Insert. */
/* */
unsigned filler : 3; /* Заполнитель... */
unsigned hold_state : 1; /* Включен Suspend. */
unsigned scroll_shift : 1; /* Нажат Scroll Lock. */
unsigned num_shift : 1; /* Нажат Num Lock. */
unsigned caps_shift : 1; /* Нажат Caps Lock. */
unsigned ins_shift : 1; /* Нажат Insert. */
};

```

Некоторые поля в подобной структуре могут быть безымянными; их описание состоит только из двоеточия и размера. Такие поля используются для подгонки размера структуры к задаваемому снаружи смещению. Особый случай - безымянное поле размера 0: оно воспринимается компилятором как требование выровнять следующее поле к границе слова. (Понятие "слово" в "библии" не конкретизируется.) В Турбо Си можно установить выравнивание к слову (значение Word переключателя Alignment в меню Options/Compiler/Code generation; опция командной строки -a) - при этом адреса объектов могут быть лишь четными, или к байту (значение Byte переключателя Alignment... в меню Options/Compiler/Code generation; опция командной строки -a-) - при этом адреса объектов могут быть любыми.

Объединения - это структуры, поля которых могут занимать одно и то же место в памяти. Общий синтаксис описания объединения таков:

```

union <имя структуры> {
    <тип 1> <имя поля 1>;
    <тип 2> <имя поля 2>;
    .....
    <тип n> <имя поля n>;
};

```

Приведем пример - из DOS.H, библиотечного файла Турбо Си:

```

struct WORDREGS {
    unsigned int ax, bx, cx, dx, si, di, cflag, flags;
};
struct BYTEREGS {
    unsigned char al, ah, bl, bh, cl, ch, dl, dh;
};

```

```
union REGS {
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

Ответственность за интерпретацию полей объединения целиком ложится на программиста. Если необходимо знать, что именно находится в объединении, то программист сам должен ввести соответствующее поле, как это сделано в нижеследующем примере:

```
struct FBR_data {
    int sex; /* 'м' - 1, 'ж' - 2 */
    char name[25]; /* Имя */
    char surname[30]; /* Фамилия */
    union {
        <данные, специфичные для мужчин>;
        <данные, специфичные для женщин>;
    } common;
};
```

В объединении можно, конечно, объявить и битовые поля. При этом, однако, правила, касающиеся безымянных полей, конечно, не применимы. Так, для объединений

```
union unit3 {
    unsigned h1 : 3;
    unsigned h2 : 3;
    unsigned h3 : 3;
    : 0;
    : 3;
} unit3;
```

и

```
union unit4 {
    unsigned h1 : 3;
    unsigned h2 : 3;
    unsigned h3 : 3;
    : 3;
} unit4;
```

имеет место равенство

```
sizeof(union unit3) = sizeof(union unit4) = 1
```

Типы `struct` и `union` позволяют однажды выписать громоздкую часть описания и использовать ее несколько раз. Описать структуру или объединение, содержащие вхождение самих себя, нельзя, но структура или объединение могут содержать ссылку на самих себя.

В приведенных примерах объявление структуры или объединения сопровождалось введением имени соответствующего типа. Это, конечно, вовсе не обязательно - достаточно повторять описание структуры при объявлении переменной. Так, вместо введения структуры `complex` и следующего за ним описания переменных мы могли бы написать:

```
struct {
    double real, img;
} z;
struct {      double real, img;      } *pz = &z;
```


правда, такая инициализация вызовет предупреждение компилятора "Suspicious pointer conversion" ("Подозрительное преобразование указателя"), однако работать все будет правильно.

Бывают, однако, случаи, когда при описании структуры или объединения невозможно обойтись без имени типа. Такая ситуация возникает в тех случаях, когда структура или объединение должны содержать ссылку на себя.

Пусть, например, в программе имеется описание структуры:

```
struct tnode { /* Представление бинарного дерева */
               /* в виде двусвязного списка */
    char tword[20];
    int count;
    struct tnode *left; /* Ссылка на левое поддерево */
    struct tnode *right; /* и на правое поддерево */
};
```

она содержит массив из 20 символов, целое и два указателя на подобные же структуры. Для объявления поля, являющегося ссылкой на структуру, необходимо придумать имя типа и использовать его при спецификации этого поля (вариант с введением пользовательского типа с помощью typedef см. ниже). При наличии такого описания объявление

```
struct tnode s, *sp;
```

говорит, что *s* - структура упомянутого вида, а *sp* - указатель на нее. После таких описаний выражение

```
sp->count
```

ссылается на элемент *count* в структуре, на которую ссылается указатель *sp*, а выражение

```
s.left
```

относится к указателю на левое поддерево в структуре *s*;

```
s.right->tword[0]
```

адресует первый символ массива *tword* в правом поддереве в *s*.

А вот как может выглядеть тот же вариант, если мы захотим использовать typedef:

```
typedef struct tnode *NODE_PTR;
typedef struct tnode {
    char tword[20];
    int count;
    NODE_PTR left; /* Ссылка на левое поддерево */
    NODE_PTR right; /* Ссылка на правое поддерево */
} NODE;
```

Мы могли бы объединить описание структуры с объявлениями переменных:

```
struct tnode { /* Представление бинарного дерева */
               /* в виде двусвязного списка */
    char tword[20];
    int count;
    struct tnode *left; /* Ссылка на левое поддерево */
    struct tnode *right; /* Ссылка на правое поддерево */
} s, *sp;
```

Или так:

```

NODE s;
NODE_PTR sp;

```

Последний вариант предпочтительнее!

11.6. МОДИФИКАТОРЫ CONST И VOLATILE

Модификатор `const` (определен в ANSI Си) говорит компилятору, что значение объекта не должно изменяться, поэтому компилятор будет считать ошибкой всякое присваивание переменной, в описании которой имеется этот модификатор. Это, конечно, не значит, что его невозможно “обмануть”. Пусть, например, в программе встретилась строка

```
const i;
```

прямая попытка употребить идентификатор `i` слева от знака присваивания встретит отпор со стороны компилятора. Однако более изощренное именуемое выражение

```
*(int *)&i = 11;
```

не встретит возражений с его стороны. Если с этим модификатором будет описан указатель, то всякое присваивание значения объекту, именуемому этим указателем, компилятор считает ошибкой.

Обратим здесь внимание на использование модификатора переменной `const`. В объявлениях

```

const int i = 5;
int * const ip = &i;

```

мы говорим, что `i` и `ip` должны сохранять постоянное значение, и потому изменить его “штатными” средствами компилятор нам не позволит: такие операторы, как `i=6` или `ip=NULL`, вызовут сообщение об ошибке “**Cannot modify a const object in function ...**” (“Не могу изменить значение *const* объекта в функции ...”). Более того, компилятор отследит, что в `ip` содержится адрес `i`, значение которого нельзя менять, и потому обругает и оператор `*ip = 6`. Однако, если мы инициализируем `ip` адресом другой переменной, не объявленной с модификатором `const`, то оператор `*ip=6` не вызовет возражений компилятора. Объявление

```
int i=5, *const ip = &i;
```

говорит компилятору лишь о том, что постоянное значение должен сохранять объект, адресуемый указателем `ip`. Поэтому операторы `i=6` и `ip=NULL` не вызовут у него возражений, а в случае

```

int i=1, j=2;
int *const qi, *const qj=&j; /* неизменяемым объявляется */
                          /* адресуемый объект */
const int *pi, *pj=&j;      /* неизменяемым объявляется */
                          /* сам указатель */

```

```

void main(void)
{
    qi = qj;
    pi = pj;
}

```

```

    *qi = *qj;
    *pi = *pj;
}

```

сообщение компилятора об ошибке вызовут первый и последний операторы.

Интерпретация модификатора **volatile** (также определен в ANSI Си) в определенном смысле противоположна **const**. Он сообщает компилятору, что объявляемый объект может измениться, причем это может произойти самым неожиданным образом, так что при операциях с значением объекта нельзя использовать его значение, которое ранее попало на регистр. Это значит, в частности, что такая переменная никогда не будет сделана регистровой.

В документации по Турбо Си в этом месте для примера приведен фрагмент программы, в котором значение переменной изменяется в подпрограмме прерывания:

```

volatile int ticks;
interrupt timer()
{
    ticks++;
}
void wait(int interval)
{
    ticks = 0;
    while (ticks < interval);
}

```

Если удалось добиться, чтобы функция **timer()** активизировалась по прерыванию от часов (в Турбо Си имеются соответствующие средства, о них мы собираемся рассказать в одном из следующих выпусков серии), то функция **wait()** задержит исполнение процесса на время, заданное ее параметром **interval**. Модификатор **volatile** необходим здесь из-за того, что при оптимизации компилятор может опустить операцию загрузки значения **ticks** из памяти, поскольку внутри цикла оно не изменяется, так что цикл никогда не закончится.

Предопределенные (псевдо)переменные.

Символы

<u>_AH</u>	<u>_AL</u>	<u>_AX</u>	<u>_BH</u>
<u>_BL</u>	<u>_BX</u>	<u>_CH</u>	<u>_CL</u>
<u>_CX</u>	<u>_DH</u>	<u>_DL</u>	<u>_DX</u>
<u>_SI</u>	<u>_DI</u>	<u>_BP</u>	<u>_SP</u>
<u>_ES</u>	<u>_SS</u>	<u>_CS</u>	<u>_DS</u>

задают (псевдо)переменные целого типа, которые не нуждаются в объявлении и позволяют программисту осуществлять прямой доступ к содержимому регистров. К ним, конечно, не применима операция **&** взятия адреса.

12. ОПИСАНИЯ ФУНКЦИЙ

Функция - единственное место в программе, где можно размещать операторы программы, именно поэтому они так важны. С функциями связаны основные претензии к Си, и подавляющее большинство ошибок также связано с функциями и указателями на них.

В этом разделе рассмотрено следующее:

- объявления, прототипы и определения функций;
- способы изменения значений объектов, находящихся вне функций;
- функции с переменным числом аргументов;
- работа с указателями, содержащими адреса функций;
- функция `main()`.

Кажется, здесь упомянуто все, что нужно для использования функций.

12.1. ОСНОВНЫЕ СВЕДЕНИЯ О ФУНКЦИЯХ

Всякий идентификатор, за которым следует открывающая круглая скобка, компилятор рассматривает как идентификатор функции.

По умолчанию компилятор Си считает, что функция возвращает целое значение. Но жизнь устроена так, что нам нужны функции, возвращающие вещественные значения, структуры, массивы или даже указатели на другие функции.

Обычный выход в такой ситуации - указание типа функции при ее определении. Выглядит это так:

```
/* Определение функции: поскольку тип пропущен, */  
long LongInt (int a, long b) /* подразумевается int. */  
{  
    /* тело функции, возвращающей значение длинного целого типа */  
}
```

Если мы вызываем функцию в том же файле, где находится этот текст, причем после него, то этого достаточно. Если же вызов находится в другом файле или текстуально ранее определения, то необходимо сообщить компилятору по крайней мере тип возвращаемого функцией значения, чтобы он сгенерировал коды, принимающие именно этот тип, который отличен от `int`. Для этого нужно объявить функцию:

```
long LongInt ();
```

Как видите, такое объявление - в главе 10 оно названо декларацией - похоже на заголовок функции с той лишь разницей, что упоминание о параметрах отсутствует. Но пользоваться такой неполной информацией о функции довольно опасно. Объясним, почему.

Если компилятор судит о функции только по приведенному объявлению, он, встретив в выражении конструкцию `LongInt(i, y)`, выдаст предупреждение *"Call to function 'LongInt' with no prototype in function..."* (*"вызов функции без прототипа"*). Что произошло? Мы не сообщили компилятору ничего о параметрах функции - он не знает ни их типов, ни даже их числа. Поэтому любое обращение к объявленной таким образом функции (независимо от числа фактических параметров и их типов) будет считаться правильным. И подозрительным! Функция получит ровно столько параметров, сколько указано в каждом конкретном вызове, без всякого контроля и преобразования типов, а как она с ними будет работать - дело ее.

Иногда такая гибкость бывает даже полезна (ниже приведен соответствующий пример), однако гораздо чаще пренебрежение полным описанием функции в объявлении приводит к трудным ошибкам. Пусть, например, в качестве первого параметра в обращении к `LongInt` использована переменная, содержащая адрес целого числа (обычное дело - пропущена звездочка перед именем указателя). Обманутая функция получит какой-то адрес (причем, скорее всего, не весь адрес, а только его часть) и начнет работать с ним как с целым. Результат трудно предсказать, ошибку трудно обнаружить. Обидно, что мы никогда бы не потратили время на борьбу с ней, если бы не поленились написать прототип, который должен выглядеть так:

```
long LongInt (int,long);
```

В прототип можно вносить и имена параметров:

```
long LongInt (int a, long b);
```

разумные имена параметров часто помогают догадаться об их смысле, т.е. повышают "самодокументированность" программы.

Правила хорошего тона диктуют полное прототипирование всех используемых функций. Выше продемонстрирован современный стиль определения функции (рекомендованный ANSI): имена формальных параметров сопровождаются указанием их типов. Встречается и старый стиль:

```
long LongInt (a,b)
```

```
int a;
```

```
long b;
```

```
{
```

```
...
```

```
}
```

Как видите, он отличается тем, что в заголовке функции указываются лишь имена формальных параметров, а их типы - после заголовка. Предупредим, что если функция не прототипирована, то старый стиль снова приведет к предупреждению компилятора *"Call to function <имя функции> with no prototype..."* (если, конечно, вы его не отключили, чего мы настоятельно рекомендуем НЕ делать). Полное прототипирование помогает компилятору обнаружить неправильные обращения к функции. Кроме того, иногда оно позволяет программисту не заботиться о преобразовании параметров - компилятор сделает это за него. Так, в выражении `LongInt(6,7)` число 7 компилятор преобразует к виду `long int`.

Ясное дело, если функция возвращает что-нибудь посложнее, чем целое, пусть даже и длинное, то и перед именем функции в ее определении и прототипе будет стоять что-нибудь посложнее, чем `long`. Зато если функция вообще ничего не возвращает, перед ее именем нужно написать волшебное слово `void` (по-английский произносится с ударением на первом

слоге - ни разу не слышал от русских программистов это слово правильно!). Это слово вообще используется в Си направо и налево: функция ничего не возвращает - ставь перед ее именем `void`, у функции отсутствуют параметры - ставь вместо них в скобках его же, хочешь определить указатель "ни на что" - ставь вместо типа его же (ну и где-нибудь звездочку, разумеется). За `void` приходится и расплачиваться: в `void`-функции нельзя использовать `return` с выражением, а с `void`-указателем нельзя производить арифметические действия - можно только сравнивать и присваивать.

Несмотря на всю свою любовь к языку Си (это не шутка!), мы должны признаться, что расшифровка достаточно хитрого объявления часто требует незаурядных криптографических способностей. Что, например, может означать следующее объявление (этот пример взят из работы Б.Андерсена "Синтаксис типов в языке программирования Си", см. "Языки программирования Ада, Си, Паскаль", М.: Радио и связь, 1989, стр. 259- 267):

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

Ответ будем искать методом последовательных приближений. Прежде всего, поскольку за `f` следует открывающая круглая скобка, `f` - функция. Далее, перед `f` стоит звездочка, значит, эта функция возвращает указатель. Пара квадратных скобок, завершающая объявление, дополнительно сообщает, что возвращается не просто указатель, а указатель на массив. Наконец, поскольку в начале объявления стоит `int`, функция возвращает указатель не просто на массив, а на массив целых чисел. Внешние круглые скобки вокруг функции чрезвычайно важны - если бы они отсутствовали, мы объявили бы массив указателей на целое!

Мы видим, что разбор кем-то сделанного объявления потребовал от нас достаточных усилий. Какие же усилия потребуются, чтобы написать такое самостоятельно!

Конечно, в данном конкретном случае мы можем вывернуться, основываясь на понимании Си. Заметим, что массив и указатель на его начало - с практической точки зрения одно и то же. Например, если у нас имеется объявление

```
int *array;
```

то выражение вида `array[11]` вполне корректно. Из-за этого в сложных случаях почти никогда не возникает необходимость описывать сложные объекты как массивы. В частности, вышеприведенное объявление можно записать гораздо проще:

```
int **x();
```

После этого выражение `x(99)[7]` вполне допустимо и его тип - указатель на `int`.

Однако всегда ли нам удастся выйти из положения так дешево? Как, например, написать объявление функции, возвращающей в качестве своего значения указатель на массив указателей на функции, возвращающие в качестве своего значения значение символьного типа (пример из той же статьи)? Совершенно ясно, что в объявлении появится целая россыпь звездочек, но где они должны стоять?

Все, в общем, не так уж страшно, если последовательно применять `typedef` и поработать мозгами. Начнем с конца - функция, возвращающая значения символьного типа, объявляется чрезвычайно просто:

```
char f();
```

поэтому мы можем запросто ввести соответствующий тип:

```
typedef char RET_CHAR_PTR();
```

Слово `RET_CHAR_PTR` у нас теперь зарезервировано для указателей на функции, возвращающие значение символьного типа и если мы напишем

```
RET_CHAR_PTR qq;
```

компилятор будет знать, что

`qq` - это указатель на функцию, возвращающую символ. Далее, мы хотим определить массив таких указателей - прекрасно, можем написать так:

```
RET_CHAR_PTR *qq;
```

либо так:

```
RET_CHAR_PTR qq[];
```

Второй вариант мы отвергнем по тем же причинам, что и раньше, и снова используем `typedef`:

```
typedef RET_CHAR_PTR *RET_CHAR_PTR_ARRAY;
```

Теперь слово `RET_CHAR_PTR_ARRAY` обозначает массив указателей на функции, возвращающие символы, и мы можем окончательно переменную нужного нам типа одним из двух способов:

```
RET_CHAR_PTR_ARRAY *x;
```

или

```
RET_CHAR_PTR_ARRAY x[SIZE];
```

Рекомендуем сравнить это с вариантом, приведенным в цитированной статье:

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

и восхититься.

Продемонстрируем, как все это работает. Нижеследующая программа выведет на экран "сообщение" 1 2 3 4.

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

```
typedef char RET_CHAR_PTR(void);
```

```
/* Укажем точный прототип - отсутствие параметров */
```

```
typedef RET_CHAR_PTR *RET_CHAR_PTR_ARRAY;
```

```
char a1(void)
```

```
{
```

```
    return '1';
```

```
}
```

```
char a2(void)
```

```
{
```

```
    return '2';
```

```
}
```

```
char a3(void)
```

```
{
```

```
    return '3';
```

```
}
```

```
char a4(void)
```

```
{
```

```
    return '4';
```

```
}
```

```
void main(void)
```

```
{
```

```
    RET_CHAR_PTR_ARRAY xx[] = {a1,a2,a3,a4,NULL};
```

```
    RET_CHAR_PTR_ARRAY *x;
```

```
    for(x = xx;*x;x++)
```

```

    printf(" %c",(*x)());
}

```

По-видимому, в заключение нужно сказать, что функции в Си, конечно, допускают прямые и косвенные рекурсивные вызовы. Приведем здесь пример, уже навязший в зубах, - вычисление факториала:

```

#include <stdio.h>
double factorial(int n)
{
    return n > 1 ? factorial(n-1)*n : 1.0;
}
void main(void)
{
    int n=1;
    printf("\nВычисление факториала");
    printf("\nЧтобы выйти из программы, введи 0");
    while(1) {
        printf ("\nВведи n:");
        scanf("%d",&n);
        if(n)
            printf("%d! = %lf",n,factorial(n));
        else
            break;
    }
}

```

Не рекомендуем, однако, пользоваться этой программой для вычисления факториала: если даже все будет посчитано правильно, то при достаточно большом n можно получить сообщение "Stack overflow!" (*переполнение стека*). Конечно, чтобы вычислять факториал, нужно пользоваться итеративным алгоритмом, причем не таким бесхитростным.

12.2. АРГУМЕНТЫ ФУНКЦИИ

Аргументы функций могут иметь любые нужные программисту типы. Приведем пример прототипа функции, аргументы которой - структуры:

```

struct point {double x,y;};
double Dist (struct point x, struct point y);

```

Обратим внимание на то, что в "библии" не разрешается передавать ни структуру, ни объединение в качестве параметра функции - только ссылку на нее. В ANSI и Турбо Си это разрешено, однако компилятор Турбо Си обязательно предупреждает об этом. Более того, функция может и возвращать структуру, как в нижеследующем примере:

```

struct point {double x,y;};
struct point Reverse(struct point a)
{
    struct point r;
    r.x = a.y;
    r.y = a.x;
    return r;
}

```



```

void main(void)
{
    struct point r,
        a = {5,6}; /* компилятор преобразует */
                    /* 5 и 6 в double */
    r = Reverse(a); /* возможность, отсутствующая в */
                    /* "библии": присваивание структур */
                    /* ... продолжение программы ... */
}

```

Здесь компилятор предупредит программиста: "Structure passed by value" (не удивляйтесь - подобное предупреждение возникнет, даже если параметром будет не структура, а объединение!). Если какой-то функции нужно передать не самое структуру, а ссылку на нее, следует, конечно, использовать операцию '&', описанную ниже.

Удобно описывать пользовательские типы с помощью typedef - в прототипах функций при этом достаточно указывать только идентификатор типа, данный при его определении. Чтобы поупражняться, перепишем предыдущую иллюстрацию:

```

typedef struct {double x,y} POINT;
double Dist (POINT x, POINT y);
POINT Reverse(POINT);

```

Теперь прототип функции не содержит ключевого слова struct. Подобное "упрятывание" описания типа чрезвычайно полезно. Вспомним, например, про структуру FILE, которой мы пользуемся при работе с файлами. В разных компиляторах эта структура устроена по-разному, однако до тех пор, пока мы не лезем внутрь структуры, наши программы вполне транспортабельны.

Если у функции отсутствуют параметры, то явно сказать об этом можно словом void:

```

int KeyPressed(void)
/* Оператор ch = KeyPressed() полностью эквивалентен */
/* оператору ch = bioskey(0), где bioskey() - */
/* библиотечная функция Турбо Си с прототипом */
/* в файле bios.h. */
{
    int ch;
    if((ch = getch()) == 0)
        /* Нажата одна из спец. клавиш */
        return getch() << 8; /* Вернем ее скэн-код */
                               /* в старшем байте */
    return ch; /* Вернем ASCII-код клавиши */
               /* в младшем байте */
}

```

Аргументы функции обязательно копируются (в стек), так что их вполне можно изменять в ее теле, используя их как обычные переменные: unsigned StrLen(char *string)

```

{
    int len=0;
    while(*string++) len++;
    return len;
}

```

Ну а как же быть, если нам нужно изменить какой-либо параметр? Конечно, при помощи указателя! Поупражняемся еще раз и перепишем нашу функцию `Reverse()` так, чтобы она меняла структуру, являющуюся ее параметром:

```
struct point {double x,y;};
void PointReverse(struct point *a)
{
    double r;
    r = a->y;
    a->y = a->x;
    a->x = r;
}

void main(void)
{
    struct point
        a = {5,6}; /* компилятор преобразует */
                  /* 5 и 6 в double */
    PointReverse(&a);
    /* ... продолжение программы ... */
}
```

Понятное дело, если параметром функции является массив, то в стек копируется адрес его начала. Это позволяет нам изменять массивы в функциях, не применяя к ним (запрещенную!) операцию `&`:

```
#include <stdio.h>
#define SIZE 100
/* Объявим слово numbers именем нового типа */
typedef double numbers[SIZE];
void init_x(numbers x,int size)
/* Присвоим элементам массива начальные значения - */
/* "квази-псевдослучайные" числа между 0 и 1 */
{
    while(size--)
        x[size] = (unsigned)(size * 271829L + 314159L)
                  / (double)0xFFFFF;
}
long double mean(numbers x, int size)
/* Подсчитаем среднее */
{
    long double sum=0;
    int i;
    for (i=0;i<size;i++)
        sum += x[i];
    return sum/size;
}

void main(void)
{
    int i;
    char *fmt = "    %lf";
```

```

numbers x;
init_x(x,SIZE); /* Присвоим начальные значения */
for(i=0;i<SIZE;i++) { /* Вывод на экран*/
    fmt[0] = i % 5 ? ' ' : '\n'; printf(fmt,x[i]);
}
/* Вычислим и выведем среднее */
printf("\n mean = %Lf", mean(x,SIZE));
}

```

Приведенный в функции `init_x()` алгоритм - почти готовый датчик псевдослучайных чисел. Конечно, перед использованием его нужно проверить; соответствующие алгоритмы описаны во втором томе великолепной книги Д.Кнута "Искусство программирования для ЭВМ".

12.3. ФУНКЦИИ С ПЕРЕМЕННЫМ ЧИСЛОМ АРГУМЕНТОВ

В Си имеется возможность писать функции с переменным числом аргументов. Необходимые для этого средства находятся в файле `stdarg.h`. Имеется ограничение: у функции должен быть по крайней мере один фиксированный аргумент. Другое ограничение связано с типизацией: нам все равно придется как-то узнавать типы аргументов. Опишем общую схему.

1. Включите в программу файл `stdarg.h`.
2. Напишите прототип функции. Он имеет следующий вид:
`<тип> <имя> (<список фиксированных аргументов> , ...)` .
3. Макрос `va_list`, определенный в файле `stdarg.h`, используется для объявления указателя на список переменной части аргументов:
`va_list <имя рабочего указателя>;` .
4. Объявленному указателю присваивается начальное значение с помощью макроса `va_start`, определенного в том же файле. Оператор имеет следующий вид:

```

va_start( <имя рабочего указателя> ,
          <имя последнего из фиксированных аргументов> ) .

```

В результате рабочий указатель будет содержать адрес первого аргумента из переменного списка. В Турбо Си второй аргумент этой функции может быть пустым, однако мы категорически не рекомендуем использовать подобные случайности реализации.

5. Доступ к аргументу осуществляется с помощью функции `va_arg` с двумя аргументами: первый из них - текущее значение рабочего указателя (содержащего адрес очередного аргумента), второй - тип этого аргумента. В качестве побочного эффекта функция `va_arg` меняет значение рабочего указателя - он начинает указывать на следующий аргумент.

6. Чтобы прекратить чтение списка переменной части аргумента, используется функция `va_end`, параметр которой - рабочий указатель.

Повторим, что, хотя в Турбо Си все эти `va...` реализованы в виде макросов, мы настоятельно рекомендуем пользоваться приведенной стандартной схемой и не раскрывать их самостоятельно: поскольку в других компиляторах вся кухня может быть совсем другой, вы лишитесь транспортабельности программы.

В качестве примера использования всего этого хозяйства мы построим функцию `imax()`, которая находит максимум последовательности целых чисел; количество чисел задается первым параметром. Функция возвращает найденный максимум.

```
#include <stdarg.h>
#include <stdio.h>
int imax(int n, int i1,...)
/* n - количество чисел, i1 - первое из них, */
/* далее следуют остальные */
{
    int result = i1,i;
    va_list(ints); /* Объявление рабочего указателя */
    va_start(ints,i1); /* Теперь ints указывает на */
                        /* первый аргумент из */
                        /* переменного списка */
    while(--n)
        if (result < (i=va_arg(ints,int))) result = i;
    va_end(ints); /* В Турбо Си это не обязательно */
    return result; /* Вернем результат */
}

void main(void)
{
    int i1=1,i2=2,i3=3,i4=4,i5=5;
    /* Передадим числа через переменные */
    printf("\nМаксимум из (%d %d %d %d %d) равен %d",
           i1,i2,i3,i4,i5,
           imax(5,i1,i2,i3,i4,i5));
    /* Передадим числа прямо в аргументах */
    printf("\nВторой максимум равен %d",
           imax(5,10,22,3,44,15));
}
```

12.4.УКАЗАТЕЛИ

Напомним, что идентификатор с открывающей скобкой за ним компилятор интерпретирует как имя функции. Просто имя функции, без скобки, дает нам адрес этой самой функции (точнее, адрес точки входа в нее). С таким адресом мы можем делать все, что позволено делать с объектами ссылочного типа. Чаще всего их используют для двух целей.

Можно завести указатель и, присваивая ему адреса разных функций, обращаться к разным функциям из одного места программы. Сходу звучит немного дико, неправда ли? Однако ситуаций, когда без подобных трюков жизнь становится довольно кислой, - тьма-тьмушая; один из примеров - модное сейчас объектно-ориентированное программирование. Мы, конечно,

не будем тут наспех объяснять, что это такое. Вместо этого приведем пример.

В этом примере заодно демонстрируем давно обещанное: здесь игнорируются предупреждения компилятора.

```
#include <stdio.h>
int plus(int x, int y)
{
    return x+y;
}
int minus(int x, int y)
{
    return x-y;
}
int sign(int y)
{
    return -y;
}
void main(void)
{
    int (*fptr)(int,...); /* fptr - указатель ! */
    fptr = plus;
    printf("\n %d + %d = %d", 5,6,fptr(5,6));
    fptr = minus;
    printf("\n %d - %d = %d", 5,6,fptr(5,6));
    fptr = sign;
    printf("\n -%d = %d", 6,fptr(6));
}
```

В результате компиляции этого примера будет выдано сообщение **"suspicious pointer conversion in function main"** (*"подозрительное преобразование указателя"*) для каждой строки, в которой переменной `fptr` присваивается в качестве значения идентификатор функции.

Если убрать из описания переменной `fptr` упоминание о параметрах, т.е. написать во второй строке `int (*fptr)()`, то это сообщение изменится на следующее **"call to a function with no prototype in function main"** (*"вызов функции без прототипа"*).

Нам не удалось придумать такой способ написать эту программу, который позволил бы и обращаться через указатель к функциям с разным числом аргументов, и ублажить компилятор, чтобы он не предупреждал нас о возможной ошибке. Впрочем, мы ведь нигде и не утверждали, что предупреждение - обязательно ошибка: просто нужно всегда обращать внимание на предупреждения компилятора и игнорировать их лишь при полным понимании своей правоты и неизбежности предупреждения.

Другое применение имени-адреса - передать его в качестве фактического параметра в функцию.

Вот и пример.

Здесь мы хотим вычислять интеграл функцией `integral()`, причем одним из параметров ее является подынтегральная функция. Способ, стандартный для Паскаля, который состоит в том, чтобы завести "фиктивную" функцию с фиксированным именем, единственное назначение которой - вызывать истинную подынтегральную функцию, мы с негодованием отвергаем. В частности, потому, что в одной программе нам может понадобиться

посчитать интеграл для нескольких функций. Конечно, можно было бы ввести в эту фиктивную функцию переключатель, но все эти убогие хитрости... Мы пойдем иным путем - будем передавать функции `integral()` имя в качестве одного из параметров!

```
#include <stdio.h>
double lin(double x) /* Подинтегральная функция */
{
    return 2*x+1;
}
double integral(double left, double right,
                double f(double x), int n)
/* left и right - соответственно левая и правая границы */
/* отрезка интегрирования; */
/* f(x) - подинтегральная функция; */
/* n - количество кусочков, на которые делим отрезок. */
/* Использован школьный метод - разбиваем отрезок, */
/* строим прямоугольники, складываем их площади. */
{
    double x, step, sum=0, fx;
    if (left < right) {
        step = (right - left)/n;
        for(x=left+step; n--; x+=step) {
            fx = f(x);
            sum += fx;
        }
    }
    return sum*step;
}
void main(void)
{
    /* Обратите внимание на то, что имя */
    /* подинтегральной функции пишется без */
    /* всяких скобок и прочих атрибутов! */
    printf("\nintegral = %lf", integral(0,1,lin,10));
}
```

Конечно, мы надеемся, что если уж действительно возникнет необходимость вычислить интеграл, вы найдете более разумный алгоритм.

12.5. ФУНКЦИЯ MAIN()

Мы уже сталкивались с функцией `main()` в приводимых примерах. Такая функция в программе может быть только одна, исполнение программы начинается с нее; то же самое по-другому: адрес точки входа в функцию `main()` фиксируется как начальный адрес программы.

В приведенных примерах эта функция была определена как `void`, т.е. не возвращающая значение, и вместо списка аргументов тоже стояло это волшебное слово. Однако часто нам нужно вернуть - операционной системе или другой вызывающей программе - некоторый код завершения программы. В этом случае типом возвращаемого функцией `main()` значения следует сделать `int`.

Кроме того, нам часто нужно при запуске программы передать ей какие-то управляющие параметры - скажем, имя файла данных. Для таких ситуаций в MS DOS имеется командная строка. Чтобы получить доступ к этой строке из программы на Си, нужно по-особому написать заголовок функции main(): при запуске программы MS DOS передаст ей из командной строки сами параметры и их количество. Заголовок должен выглядеть так:

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

При таком заголовке в argc будет помещено количество параметров, а в argv[] - указатели на строки, содержащие эти параметры. Напомним, что каждое слово, не содержащее пробелов, будет считаться отдельным параметром. Последний элемент массива argv[] - обязательно NULL (он параметром не считается!), первый (в DOS 3.0 и выше) - полное имя запускаемой программы: имя директория (в других переводах - каталога), где находится файл, содержащий программу, и имя самого файла.

Чтобы не погрязнуть в достаточно тривиальных объяснениях, приведем пример - напомним аналог программы ASK из знаменитейшего Norton Commander'a. На входе у программы ASK два параметра - строка (вопрос, который нужно задать пользователю) и совокупность букв, допустимых в качестве ответов: если пользователь нажимает клавишу с неуказанной буквой, программа "пищит". В вызывающую программу ASK передает порядковый номер нажатой буквы в заданной совокупности (конечно, из числа допустимых). Пусть, например, программа запущена так:

```
ASK Готов?, дн
```

Тогда на нажатие любой клавиши, кроме 'д' и 'н', программа запищит; если нажата 'д', то в вызывающую программу она передаст значение 1, а если нажата 'н', то в вызывающую программу она передаст 2. Конечно, если нужно, чтобы подсказка состояла из нескольких слов, ее следует окаймить кавычками:

```
ASK "Ты готов, наконец?", дн
```

Приведем текст программы, которая реализует эти действия. Отличия от ASK: совокупность возможных ответов не нужно отделять запятой - достаточно пробела, и если пользователь нажмет клавишу ESC, то программа возвратит 0.

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

/* определим код клавиши ESC */
#define ESC 27
char *usage = "вызов: ASK \"подсказка\" список клавиш";
int main(int argc, char *argv[])
{
    int i, len, ch;
    if (argc != 3)
        /* Если командная строка          */
        /* содержит не два аргумента,        */
        /* вывести правила пользования.      */
        printf("%s", usage);
    else {
        /* При вызове ASK "Ты готов?" дн      */
        /* argc = 3, причем:                   */
        /* argv[0] содержит имя программы     */
```

```
/* argv[1] содержит "Ты готов?" */
/* argv[2] содержит строку "дн" */
/* argv[3] содержит NULL */
printf("%s (%s) ",argv[1],argv[2]);
len = strlen(argv[2]);
while(1) {
    if((ch = getch()) == ESC)
        return 0; /* Нажата клавиша ESC - вернем 0 */
    for(i=0;i<len;i++)
        /* Без явного преобразования типа */
        /* сравнение не будет работать! */
        if ((char)ch == argv[2][i])
            return i+1; /* Вернем номер буквы */ putchar('\a');
    /* Нажато не то - погудим! */
}
}
```


13. ОПЕРАЦИИ И ВЫРАЖЕНИЯ

13.1. ПРЕДВАРИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

Выражение - это, в конечном счете, единственный способ сказать программе, что именно необходимо сделать. Выражения в Си, как и в любом другом языке, состояются из имен объектов, определенных программистом, "разбавленных" знаками операций. Значение выражения присваивается объекту, стоящему слева от знака присваивания, который в Си обозначается символом '='. Уместно, по-видимому, сразу сказать, почему мы говорим здесь об "объектах", а не о переменных, что выглядело бы гораздо естественнее.

Дело, конечно, в том, что слева от знака присваивания может в свою очередь стоять выражение, задающее адрес области памяти, где должно быть размещено полученное справа значение (в более "теоретической" англоязычной литературе по Си здесь был бы употреблен термин L-value, что, по-видимому, точнее всего перевести как "адресующее значение"). Объекты, определенные программистом, должны быть описаны заранее - в начале одного из охватывающих блоков, в котором они используются. Конечно, каждому типу объектов соответствует свой "естественный" набор операций.

Старшинство операций, определяющее правила вычисления выражений, описано в параграфе 13.13. Все, что не охватывается этими правилами и связано с порядком вычисления, остается в стандарте языка неопределенным. В частности, компилятор "оставляет за собой право" вычислять подвыражения в порядке, который он сочтет наиболее эффективным, несмотря даже на то, что выражение включает побочный эффект. Выражения, включающие коммутативные и ассоциативные операции (операции -, +, &, |, ^), компилятор может произвольно перепорядочить даже при добавлении скобок. Фиксировать определенный порядок вычислений можно явным сохранением промежуточного результата.

Обработка переполнения и контроль деления при вычислении выражения, конечно, зависят от машины, но и от средств, предоставляемых компилятором. В Турбо Си необходимые средства имеются, но мы их здесь не рассматриваем, поскольку они реализуются не средствами языка, а средствами стандартных библиотек.

Некоторые операции требуют операндов определенного вида. Вид операнда обозначается одной из следующих букв:

- e - любое выражение;
- v - любое выражение, ссылающееся на объект, которому может быть присвоено значение; такие выражения выше названы адресующими.

Тип операнда указывается префиксом. Например, *ie* обозначает произвольное выражение целого типа. Возможны следующие префиксы:

- i - выражение целого типа;
- a - выражение арифметического типа (целое число, символ или число с плавающей точкой);

- p - выражение ссылочного типа;
- s - структура или объединение;
- sp - ссылка на структуру или объединение;
- f - функция;
- fp - ссылка на функцию.

Обозначение smet используется для элементов структуры или объединения.

Если в выражение участвует несколько операндов, то они отличаются номерами, например: ae1 + ae2.

13.2. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

К арифметическим операциям относятся четыре арифметических действия и операция '%' вычисления остатка от деления, применимая лишь к целым операндам. Кроме того, мы здесь же опишем унарные операции '+' и '-', а также '++' и '--'.

Аддитивные операции.

Аддитивные операции выполняются слева направо, при этом выполняются обычные арифметические преобразования. С каждой операцией связаны и некоторые другие возможности.

<аддитивное выражение> ::=

ae1 + ae2 | pe1 + ie1 | +ae |

ae1 - ae2 | pe1 - ie1 | pe1 - pe2 |

-ae

Результат операции '+' есть сумма операндов. К указателю на объект в массиве можно добавлять любое значение целого типа, которое в таких случаях преобразуется к типу int; такое преобразование может вызвать предупреждение компилятора "Conversion may lose significant digits" (*Преобразование может вызвать потерю значащих цифр*). При преобразовании значение умножается на длину объекта, адресуемого указателем.

В результате получается указатель того же типа, что и исходный, но адресующий другой объект в том же массиве, смещенный соответствующим образом относительно первоначального. Так, если P - указатель на некоторый объект в массиве, то выражение P + 1 есть ссылка на следующий объект в этом массиве. Никакие другие типы комбинировать с указателями нельзя.

Операция сложения ассоциативна, и выражение с несколькими операциями сложения одного уровня может быть перепорядочено компилятором.

Результат операции '-' представляет собою разность операндов; при этом выполняются обычные арифметические преобразования. Кроме того, из указателя можно вычитать значение любого целого типа, в этом случае производятся преобразования, аналогичные преобразованиям для случая сложения.

Если вычитаются два указателя на объекты одного типа, результат преобразуется (путем деления на длину объекта) в значение целого типа, которое равняется числу объектов, находящихся между указанными объектами. В общем случае, если только речь не идет о ссылках на объекты

одного массива, такое преобразование может давать неожиданные результаты. Это происходит из-за того, что ссылки на объекты одного типа не обязательно разнятся на величину, кратную длине объекта.

Унарный плюс не меняет значения объекта, на который указывает операнд выражения, унарный минус меняет его на противоположный. Примеры.

```
{
    int i,j;
    long arr[100];
    long *p;
    i=-2; /* к константе 2 применен унарный минус */
    j=i; /* все равно что j=i */
    p=arr+5; /* p - адрес 6-го элемента массива arr */
    i=j-i; /* i = 0 */
    p=p-2; /* p - адрес 4-го элемента arr */
    j=p-arr; /* j = 3 */
    i=arr[2]+*p; /* i = сумма 3-го и 4-го элементов arr;
                здесь будет предупреждение */
}
```

Мультипликативные операции.

Мультипликативные операции выполняются слева направо. При этом выполняются обычные арифметические преобразования.

<мультипликативное выражение> ::=

```
    ae1 * ae2 |
    ie1 % ie2 |
    ae1 / ae2
```

Символом '*' обозначается умножение. Это ассоциативная операция и выражения с несколькими умножениями на одном уровне компилятор может "перестраивать".

Символом '%' обозначается взятие остатка от деления первого выражения на второе. Выполняются обычные арифметические преобразования. Операнды должны быть целого типа.

Деление обозначается символом '/'. В "библии" говорится, что если делятся положительные целые числа, то дробная часть отбрасывается, если же хотя бы один из операндов отрицателен, то форма округления зависит от машины (реализации). Гарантируется, тем не менее, что при любых целых операндах справедливо равенство

$$(a/b)*b+a\%b = a$$

В Турбо Си результат целочисленного деления не зависит от знака - дробная часть всегда отбрасывается. Примеры.

```
{
    int i=8,j=3;
    float f,g;
    j=i % j; /* j = 2 */
    j=i / (-3); /* j = -2 */
    j=4*i*j; /* порядок вычисления не фиксирован */
}
```

Унарные аддитивные операции.

<унарная аддитивная операция> ::=
 ++ av | av ++ | ++ pv | pv ++ |
 — av | av — | — pv | pv —

Если знак операции стоит перед операндом, то значение адресуемого объекта изменяется и лишь после этого оно участвует в выражении. Если же знак операции стоит после операнда, то в выражении фигурирует его текущее значение, после чего оно изменяется. Значением выражения ++av является "новое" значение выражения av, значением выражения pv— является "старое" значение выражения pv и т.д.

Примеры.

```
{
  int i = 1, j=2, k;
  double x=100, y=200, z;
  k = j*i++; /* k = 2, i = 2 */
  z = x * ++y / —i; /* z = 20100, y = 201, i = 1 */
}
```

13.3. ПОБИТОВЫЕ ОПЕРАЦИИ

Побитовые операции - это операции над битовыми представлениями операндов: логические AND (&), OR (|), XOR (^) и NOT(~), а также сдвиг вправо (>>) и влево (<<). Все эти операции применимы лишь к целым или приводимым к ним операндам, так что если знак операции обозначить символом @, то для всех бинарных побитовых выражений справедливо единое представление ie @ ie. Операция NOT выглядит так: ~ie.

Поразрядная операция И (ie & ie) ассоциативна и выражение, куда она входит, может быть переупорядочено. Выполняются обычные арифметические преобразования. Результат - поразрядная функция И от операндов.

Поразрядная (исключающая, XOR) операция ИЛИ (ie ^ ie) ассоциативна, поэтому выражение, включающее такую операцию, может быть переупорядочено. Выполняются обычные арифметические преобразования. Результат - поразрядная исключающая функция ИЛИ от операндов.

Поразрядная (включающая, OR) операция ИЛИ (ie | ie) ассоциативна, поэтому выражение, включающее такую операцию, может быть переупорядочено. Выполняются обычные арифметические преобразования. Результат - поразрядная включающая функция ИЛИ от операндов.

Операции сдвига (ie1 << ie2 | ie1 >> ie2) выполняются слева направо. В обоих случаях проводятся обычные арифметические преобразования операндов, каждый из которых должен быть целого типа. Затем правый операнд преобразуется к типу int, тип результата - это тип левого операнда. В "библии" говорится, что результат операции не определен, если правый операнд отрицателен, равен или превышает размер объекта в разрядах. В Турбо Си в обоих случаях операции выполняются описанным ниже стандартным способом.

Значение выражения `ie1 << ie2` получается следующим образом: совокупность битов объекта, адресуемого выражением `ie1`, сдвигается влево на (`ie2`) разрядов; освобождающиеся разряды заполняются нулевыми битами.

При вычислении выражения `ie1 >> ie2` разряды сдвигаются вправо. Если `ie1` типа `unsigned`, то гарантируется, что сдвиг вправо - логический, т.е. освобождающиеся биты заполняются нулями. В других же случаях выполняется арифметический сдвиг, т.е. свободные разряды заполняются копией знакового разряда.

Примеры.

```
{
    signed int i=0x007f;
    int j=0x7f01;
    unsigned int k;

    j=i & j;          /* j = 0x0001 */
    j=k ^ k;          /* j = 0x0000 при любом k */
    j=0x4800 | i | j; /* j = 0x4801 */
    j=~(0xffff ^ j)   /* j = 0x4800 */
    k=j << 4;         /* k = 0x8000 */
    i=j << 4;         /* i = 0x8000 */
    j=k >> 8;         /* j = 0x0080 */
    j=i >> 8;         /* j = 0xff80 */
}
```

13.4. ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Логические операции дают в результате значение 0 (интерпретируется как ЛОЖЬ) и 1 (ИСТИНА). Любое ненулевое значение операнда интерпретируется как ИСТИНА и нуль интерпретируется как ЛОЖЬ.

Логическая операция И.

<логическое И-выражение> ::= `e1 && e2`

Операции `&&` выполняются слева направо. Такая операция даст результат 1, если оба ее операнда отличны от нуля, в других случаях результат равен нулю. В отличие от `&` при операции `&&` гарантируется вычисление слева направо; более того, второй операнд не вычисляется, если первый операнд равен нулю.

Не требуется, чтобы операнды были одного типа, но каждый должен быть одного из основных типов или указателем. Результат всегда имеет тип `int`.

Логическая операция ИЛИ.

<логическое ИЛИ-выражение> ::= `e1 || e2`

Операции `||` выполняются слева направо. Такая операция дает результат 0, если оба ее операнда равны нулю, в других случаях результат равен 1. В отличие от `|` при операции `||` гарантируется вычисление слева

направо; более того, второй операнд не вычисляется, если первый операнд не равен нулю.

Операнды не обязательно одного типа, но каждый должен быть одного из основных типов или указателем. Тип результата - `int`.

Логическое отрицание.

<логическое НЕТ-выражение> ::= `!ae | !pe`

Результатом операции логического отрицания `!` будет 1, если операнд имел значение 0, если же операнд был отличен от нуля, то результатом будет 0. Типом результата будет `int`. Эту операцию можно применять к значению любого арифметического типа или к указателю.

Примеры.

```
{
    int i,j=3,k=0;
    i= k && somefun(j); /* i=0; somefun не вызывается */
    j= i || k || j;      /* j = 1 */
    j= !(i || j);        /* j = 0 */
}
```

13.5. ОПЕРАЦИИ СРАВНЕНИЯ

Операции сравнения всегда выполняются слева направо, но этот факт мало что дает, ибо выражение `a < b < c` все равно интерпретируется на так, как принято в математике.

<выражение сравнения> ::=

```
ae1 == ae2 | ae1 != ae2 |
pe1 == pe2 | pe1 != pe2 |
ae1 <= ae2 | ae1 < ae2 |
ae1 >= ae2 | ae1 > ae2
```

Операции меньше (`'<'`), больше (`'>'`), меньше или равно (`'<='`) и больше или равно (`'>='`) дают результат 0, если указанное отношение не выполняется, и 1, если оно справедливо. Тип результата `int`. Выполняются обычные арифметические преобразования. При сравнении двух указателей (именующих значений) результат может зависеть от типа указателя (см. ниже пример для сравнения на равенство).

Операции равно (`'=='`) и не равно (`'!='`) полностью аналогичны операциям отношения, за исключением того, что их приоритет ниже. Таким образом, `a < b == c < d` дает результат 1, если неравенства одновременно выполнены или одновременно не выполнены (т.е. выражения `a < b` и `c < d` имеют одно и то же значение).

В "библии" сказано, что ссылку можно сравнивать с целым значением, но результат зависит от реализации. если только целое не есть константа 0 (в Турбо Си подобные сравнения вызовут предупреждение). Гарантируется, что если значением ссылки является 0 (в Си для этого значения имеется специальное обозначение `NULL`, которое входит в стандарт и определяется по-разному в зависимости от используемой модели памяти), то оно не указывает на какой-либо объект; при обычных преобразованиях такое значение считается нулем.

Результат сравнения двух указателей в Турбо Си может зависеть от того, как они описаны. Так, нижеприведенная программа выведет на экран слово "равны":

```
#include <stdio.h>
#include <dos.h>
void main(void)
{
    char huge *p = MK_FP(0x0000,0xC0ff);
    char huge *q = MK_FP(0x000f,0x0C0f);
    char *msg;
    msg = p == q ? "равны". "Неравны";
    printf("\n%s",msg);
}
```

И в самом деле, согласно правилам формирования адреса в IBM PC два указателя адресуют одну и ту же ячейку памяти. Если, однако, в описании указателей убрать модификатор `huge`, то на экране появится слово "Неравны".

Примеры использования операций сравнения.

```
{
    int i,j=3,k=0;
    char arr[100],*p=arr,*q=NULL;
    i = j = k; /* i=0; */
    k = -1 <= i == i <= 1; /* k!=1; выражение справа от знака */
                          /* присваивания соответствует */
                          /* "обычной" записи -1 <= i <= 1 */
    j = p == NULL || q != NULL; /* j = 0 */
}
```

13.6. ОПЕРАЦИИ ПРИСВАИВАНИЯ

Имеется несколько операций присваивания, все они выполняются справа налево. Для всех них требуется, чтобы слева находилось адресующее выражение, и тип присваиваемого выражения есть тип его левого операнда, или возможно автоматическое преобразование к типу левого операнда по правилам, описанным в параграфе 13.14. Результат операции есть значение левого операнда после присваивания. В "библии" указано, что обе части составной операции присваивания являются отдельными лексемами. Это означает, что их можно разделять произвольным числом пробелов, комментариев и концов строк. В Турбо Си это не так, и выражение вида `a+ /* комментарий */ =1` вызовет сообщение об ошибке.

<присваиваемое выражение> ::=

`v = e |`

`av += ae | pv += ie | av -= ae | pv -= ie | av *= ae | av`

`/= ae | iv %= ae | iv >= ie | av <= ie | iv &= ie | iv`

`^= ie | iv != ie`

При простом присваивании ('=') значение выражения справа заменяет собой значение объекта, адресуемого выражением слева; если оба операнда арифметического типа, то правый операнд, прежде чем выполнится присваивание, преобразуется к типу левого операнда. Согласно ANSI Си (реали-

зовано в Турбо Си) в операции простого присваивания могут фигурировать структуры или объединения.

Выражение $E1 \text{ op} = E2$ эквивалентно выражению $E1 = E1 \text{ op } (E2)$, однако $E1$ вычисляется лишь один раз. В операциях $+=$ и $-=$ левый операнд может быть указателем, в этом случае правый операнд обязательно целый и преобразуется, как объяснялось выше. Все правые операнды и все левые операнды, не являющиеся указателями, должны быть арифметического типа.

Компилятор Турбо Си позволяет присваивать адресуемое значение объекту типа `long int`, указатель произвольного типа - указателю введенного в стандарте Си типа `void`, указатель типа `void` - указателю произвольного типа. Присваивание - это просто копирование без преобразования, поэтому арифметические операции над переменной типа `long int`, первоначальным значением которой был адрес, могут при обратном присваивании привести к аварийным ситуациям. Однако гарантируется, что присваивание указателю константы `NULL` будет порождать значение, не именуемое какой-либо реальным объектом. Примеры.

```
{ float f;
  int arr[2], *iptr, i=1, j, k, l;
  f=3.14;
  i+=3; /* i=4 */
  iptr=arr+1;
  iptr+=2; /* iptr - адрес четвертого элемента arr */
  i-=2; /* i=2 */
  iptr-=2; /* iptr - адрес второго элемента arr */
  i*=i; /* i=4 */
  i/=4; /* i=1 */
  i%=10; /* i=1 */
  i<<=4; /* i=0x0010 */
  i>>=2; /* i=0x0004 */
  i|=0xf000; /* i=0xf004 */
  i^=0xff00; /* i=0x0f04 */
  i&=0x00ff; /* i=0x0004 */
  j=k=i; /* j=4, k=4, l=4 */
}
```

13.7. ОПЕРАЦИЯ “ЗАПЯТАЯ”

<выражение с запятой> ::= e1 , e2

Пара выражений, разделенных запятой, вычисляется слева направо, и значение левого выражения пропадает. Тип и значение результата есть тип и значение правого операнда. Эти операции выполняются слева направо. В контекстах, где запятая имеет специальное значение, например, в списке фактических аргументов функции или в списках инициализации, операция запятой может появляться только в скобках. Вот, например, обращение к функции, содержащее три аргумента, причем второй имеет значение 5:

```
f(a, (1=3, 1+2), c)
```


13.8. АДРЕСНЫЕ ОПЕРАЦИИ

<адресная операция> ::= &v | *pe | *fpe

Результатом унарной операции '&' является адрес операнда. Тип результата - указатель на тип операнда.

Унарную операцию '*' часто называют косвенной адресацией: выражение должно быть адресуемым и результатом является значение объекта. Тип выражения есть тип объекта, адресуемого выражением.

То же самое можно выразить по-другому. Значение объекта, адрес которого содержится в указателе, можно получить, применив операцию "разыменования", обозначаемую звездочкой (*). Банальный пример:

```
void main(void)
{
    int i, j, k;
    int *iptr;
    i = 22;
    iptr = &i; /* теперь iptr содержит адрес i */
    j = i; /* j = 22 */
    k = *iptr; /* k = 22 */
}
```

Значением выражения *fpe является функция, адресуемая указателем fpe. Пример.

```
fpe = funcname;
(*fpe)(arg1,arg2);
```

Вызов функции fe с аргументами e1, e2,...,eN осуществляется оператором fe(e1,e2,...,eN). Значением такого выражения является значение, возвращаемое функцией (если только функция не имеет тип void: обращение к значениям таких функций является ошибкой). В стандарте языка порядок, в котором вычисляются выражений e1, e2,...,eN не определен; в Турбо Си эти выражения вычисляются справа налево. Например, фактически параметрами функции try с целыми аргументами в следующем фрагменте программы

```
a=1;
try(—a,(a=a+6),a+8);
будут 6, 7 и 9.
```

13.9. ОПЕРАЦИЯ УСЛОВИЯ

<условное выражение> ::= ae ? e1 : e2 | pe ? e1 : e2

Выполнение условного выражения происходит следующим образом: сначала вычисляется первое выражение (ae или pe); если оно отлично от нуля, то результатом является значение второго выражения, иначе - значение третьего выражения. При необходимости для приведения второго и третьего выражений к общему типу выполняются обычные арифметические преобразования; иначе, если оба выражения - указатели одного типа, результат будет иметь этот общий тип; иначе - константой, и результат имеет тип этой ссылки. Вычисляется либо только второе, либо только третье выражение.

Пример.

```
#include <stdio.h>
void main (void)
{
    int i=13;
    printf(i % 2 ? "\nodd" : "\neven");
    i*=i;
    printf(" %d\n",i > 100 ? i-100 : i);
}
Программа напечатает
odd
69
```

13.10. ОПЕРАЦИИ НАД МАССИВАМИ

<Элемент массива> ::= pe[ie]

Элементы массива могут фигурировать в выражении в одном из двух эквивалентных видов: можно написать либо идентификатор массива, а за ним индекс(ы) в квадратных скобках, либо использовать выражение с указателем:

*(<идентификатор массива> + <индекс>).

Отметим, что выражение `a[i]` всегда компилируется в `*(a+i)`, поэтому в Си допустимы совсем странные конструкции, как в следующем примере.

```
void main(void)
{
    int i, k;
    int a[] = {5, 6, 7, 8, 9};
    /* мы описали массив с 5 элементами */
    i = 3;
    k = a[3]; /* k = 8 */
    k = a[i]; /* k = 8 */
    k = 3[a]; /* k = 8 */
    k = i[a]; /* k = 8 */
    k = *(a+3); /* k = 8 */
    k = *(a+i); /* k = 8 */
}
```

13.11. ОПЕРАЦИИ НАД СТРУКТУРАМИ И ОБЪЕДИНЕНИЯМИ

Выражение вида `sv.smem` используется для получения доступа к элементам структуры или объединения. Выражение `sv` должно адресовать структуру или объединение, `smem` - поле в соответствующем объекте. Выражению `(*spe).smem` полностью эквивалентно выражение `spe->smem`, которое, следовательно, является его сокращением.

Пример.

```
{ struct word {unsigned char hi,lo} wd,*wptr;
```

```

/* следующие присваивания эквивалентны */
wd.hi=wd->lo;
wd.hi=(*wd).lo;
}

```

13.12. ДРУГИЕ ОПЕРАЦИИ

Операция `sizeof` задает размер операнда в байтах (байт в самом языке не определяется) - в этом случае операнд может не быть в скобках. Например, в Турбо Си `sizeof 3` дает 2, `sizeof (4/3.)` дает 8 (скобки здесь выделяют выражение), `sizeof (4/3.l)` дает 10. Если операция применяется к массиву, то результат - общее число байтов в массиве. Размер определяется на основе описания объектов, фигурирующих в выражении. Значение такой операции семантически представляет целую константу и может использоваться везде, где требуется указывать константу. В основном они используются для работы с подпрограммами типа распределения памяти или подпрограммами ввода-вывода.

Операцию `sizeof` можно применять и к имени типа; при этом имя типа указывается в скобках. Результатом является размер в байтах объекта указанного типа.

Выражение вида (типа) `e` используется для преобразования типа выражения `e` в другой тип и называется приведением типа. Это выражение очень широко используется для преобразования типа ссылочного выражения (хотя мы настоятельно рекомендуем пользоваться этой возможностью с большой осторожностью - совет основан на богатой коллекции ошибок, допущенных нами в своих программах и найденных нами в чужих). Кроме того, оно полезно в тех случаях, когда необходимо перейти, скажем, от целого операнда к плавающему. Так, $7/2$ равняется 3, а `(float)7/2` равно 6.5. (Предостережем от возможного заблуждения: `(float)(7/2)` даст 3.0.)

13.13. СТАРШИНСТВО ОПЕРАЦИЙ

Для каждой группы операций в нижеследующей таблице приоритеты одинаковы. Чем выше приоритет группы операций, тем выше она расположена в таблице. Порядок выполнения определяет группировку операций и операндов (слева направо или справа налево), если отсутствуют скобки и операции относятся к одной группе.

Старшинство операций

Операция

() [] . ->

! ~ - ++ -- & * (type) sizeof

Порядок выполнения

слева направо

справа налево

* / %	слева направо
+ -	слева направо
<< >>	слева направо
< <= > >=	слева направо
= !=	слева направо
&	слева направо
^	слева направо
	слева направо
&&	слева направо
	слева направо
?:	справа налево
= += -= etc.	справа налево
,	слева направо

13.14. ПРЕОБРАЗОВАНИЯ

Некоторые операции в зависимости от их операндов могут вызывать преобразование значений операндов из одного типа в другой. Ниже объясняется, к чему приводят такие преобразования.

Символы и целые числа.

Всюду, где можно использовать значение целого типа, можно использовать также значения короткого целого и символьного типов. В любом случае соответствующее значение автоматически преобразуется к целому. Преобразование коротких целых к длинным всегда включает "размножение" знака, ведь целые - это величины со знаком. Будет ли происходить размножение знака при преобразовании к целому значения символьного типа, зависит от установленного режима компиляции. Если, однако, при объявлении целой или символьной переменной снабдить описание типа уточняющим модификатором `unsigned`, то при преобразовании значения переменной в целое "размножение" знака произведено не будет. Наоборот, преобразование значения переменной, описанной с модификатором `signed`, всегда будет включать "размножение" знака.

При преобразовании значения длинного целого типа в значение более короткого целого или символьного типа его старшие разряды "отсекаются": лишняя часть попросту отбрасывается.

Так как представление значений перечислимых типов в точности совпадает с представлением значений целого типа, все сказанное выше о значениях целого типа справедливо по отношению к значениям перечислимых типов. В частности, преобразование из любого целого или символьного типа в перечислимый производится автоматически, при этом никак не контролируется, входит ли в множество значений перечислимого типа преобразуемое целое.

Значения с плавающей точкой и двойная точность.

В языке Си считается, что все арифметические операции всегда выполняются с двойной точностью. Поэтому всякий раз, когда в выражении появляется действительное значение, оно преобразуется к длинному действительному добавлением нулей к мантиссе. При преобразовании длинного действительного значения к одинарному, например, при присваивании, производится округление, после чего лишние младшие разряды мантиссы отбрасываются.

Плавающие типы и целые.

Преобразование действительного значения в целое зависит, конечно, от используемой машины. В Турбо Си, в полном соответствии с "библией", результат преобразования не определен, если значение не "помещается" в память, отводимую для целых.

Преобразование целых значений в действительные не вызывает каких-либо осложнений, однако, если для результата не обеспечено соответствующее число разрядов, будет происходить некоторая потеря точности.

В Турбо Си потеря точности может произойти при преобразовании от `long int` к `float`, поскольку длина мантиссы у типа `float` равняется 7 десятичным цифрам.

Указатели и целые.

Любое целое можно складывать с указателем и вычитать из него. При этом первый операнд будет преобразован в соответствии с правилами, приведенными при обсуждении операции сложения.

Два указателя на объекты одного типа можно вычитать друг из друга. Результат будет преобразован в целое значение по правилам, приведенным при обсуждении операции вычитания.

Целые без знака.

Если один из операндов является целым без знака (`unsigned int`), а второй - "обычным" целым (`signed int`), то целое преобразуется в целое без знака и результат будет целым без знака. Его значению есть наименьшее целое без знака, совпадающее с целым со знаком по модулю 65536, т.е. 2^{16} (16 - число разрядов машинного слова IBM PC).

Если целое без знака преобразуется в длинное целое, то значение результата численно то же самое, что и целое без знака, т. е. преобразование в данном случае заключается в добавлении "слева" нулей.

Арифметические преобразования.

Для подавляющего большинства операций преобразования операндов и тип результата определяются одними и теми же правилами. Эти правила, которые в "библии" названы "обычными арифметическими преобразованиями", состоят для Турбо Си в следующем:

1. Любой операнд типа **char** или **short** преобразуется в целое в соответствии с приводимой ниже таблицей, а любой операнд типа **float** преобразуется к типу **double**.

2. Если один из операндов типа **double**, то другой преобразуется к типу **double**, и результат имеет тот же тип.

3. Если же один из операндов типа **long double**, то другой преобразуется к типу **long double**, и результат имеет тот же тип.

4. В противном случае, если один из операндов типа **unsigned long**, то другой преобразуется к тому же типу, и результат будет того же типа.

5. В противном случае, если один из операндов типа **long**, то и другой преобразуется к типу **long**, и результат будет того же типа.

6. Если все предыдущее неприменимо, и один из операндов типа **unsigned**, то и другой преобразуется к типу **unsigned**.

7. В противном случае, оба операнда - типа **int**.

Таблица преобразования объектов к типу **int**

Тип	Метод
char	размножение знака, если не установлено, что по умолчанию этот тип беззнаковый
signed char	всегда размножение знака
short	если unsigned , то unsigned int enum то же значение

13.15. КОНСТАНТНЫЕ ВЫРАЖЕНИЯ

В некоторых местах в языке речь идет о выражениях, которые должны "сводиться" к константе: после слова **case**, в границах массивов и при инициации. В первых двух случаях выражение должно включать только целые константы, символьные константы и "функцию" **sizeof**. Для объединения их в выражения могут использоваться бинарные операции: **+** **-** ***** **/** **%** **&** **|** **^** **<<** **>>** **==** **!=** **<** **>** **<=** **>=** или унарные операции **-** и **~** или "тройная" операция **?:**.

Для группирования можно использовать скобки, но нельзя обращаться к функциям.

При инициации допустима большая свобода: кроме выражений из констант (о них речь шла выше) можно использовать унарную операцию **&**, применяемую к внешним или статическим объектам и к внешним или

статическим массивам, индексированным константными выражениями. Унарная операция & может появляться и неявно, при вхождении массивов без индексов или функций. Основное правило - иницилирующее значение должно сводиться либо к константе, либо к адресу предварительно описанного внешнего или статического объекта плюс/минус константа.

14. ОПЕРАТОРЫ

Все операторы, за исключением специально оговоренных случаев, выполняются один за другим.

Оператор-выражение.

Чаще всего операторами бывают выражения; любое выражение, за которым следует точка с запятой, является оператором. Обычно операторы-выражения являются присваиваниями или обращениями к функциям.

Составной оператор или блок.

Для тех случаев, когда вместо одного оператора желательно использовать несколько, предусмотрен составной оператор (иногда его называют "блок").

В начале блока можно объявить переменные, локальные в нем. Если какие-либо из идентификаторов в таком списке уже были до этого объявлены, то внешние объявления запоминаются на время работы данного блока, а после выхода из блока они вновь вступают в силу.

Инициализация автоматических или регистровых переменных выполняется при каждом входе в блок через его начало. Сейчас можно передать управление внутрь блока (но на это не стоит ориентироваться); при этом инициализация выполняться не будет. Статические переменные инициализируются только один раз, в начале выполнения программы. Описания с атрибутом `extern` внутри блока не резервируют никакой памяти, поэтому инициализация их не допускается.

Условный оператор.

Имеется два вида условных операторов:

`if (<выражение>) <оператор1>`

и

`if (<выражение>) <оператор1> else <оператор2>`

В любом случае вычисляется `<выражение>` и выполняется `<оператор1>`, если его значение не равно нулю. Во второй записи `<оператор2>` выполняется, если значение выражения есть нуль. Как обычно, двусмысленность толкования `else` разрешается путем его сопоставления с ближайшим `if` без парного `else`.

Оператор while.

Оператор имеет такой вид:

`while (<выражение>) <оператор>`

Оператор выполняется до тех пор, пока значение выражения остается отличным от нуля. Проверка значения происходит перед каждым выполнением оператора.

Оператор do.

Оператор имеет вид:

`do <оператор> while (<выражение>)`

Оператор выполняется до тех пор, пока выражение не станет равным нулю. Проверка происходит после каждого выполнения оператора.

Оператор for.

Оператор имеет вид:

`for(<выражение1>;<выражение2>;<выражение3>)
<оператор>`

Этот оператор эквивалентен такому фрагменту программы:

```
<выражение1>  
while (<выражение2>) {  
    <оператор>  
    <выражение3>;  
}
```

Таким образом, первое выражение задает исходные установки для цикла, второе выполняет проверку, а третье выражение обычно задает приращение, оно вычисляется после каждой итерации.

Любое или даже все выражения можно опускать. Если пропущено <выражение2>, то конструкция `while` неявно заменяется на `while(1)`; если же пропущены другие выражения, то они просто опускаются из поясняющей схемы. Подчеркнем, что точки с запятыми, разделяющие выражения, опускать нельзя.

Оператор switch.

Оператор `switch` (переключатель) вызывает передачу управления на один из нескольких операторов в зависимости от значения выражения. Он имеет вид:

`switch (<выражение>) <оператор>`

При вычислении выражения выполняются обычные арифметические преобразования, но результат должен быть типа `int`. Оператор обычно бывает составным. В нем можно из операторов можно поместить одним или несколькими префиксами следующий вид:

`case <состояние>: <выражение> ;`

Константное выражение должно быть целым. Никакие две константы варианта в одном переключателе не могут быть одинаковыми. Константное выражение - это выражение, которое может быть вычислено в период компиляции. Может также встретиться (но только один раз!) префикс вида `default` :

При выполнении оператора переключателя вычисляется выражение, и его значение сравнивается с каждой из констант вариантов. Если одна из констант вариантов равна значению выражения, то управление передается на оператор, идущий за этой константой. Если ни одна из констант с выражением не совпала, но есть префикс `default`, то управление передается на оператор с таким префиксом. Если такого префикса нет и совпадения не было, то в переключателе не выполняется ни один оператор.

Если операторы, выполняемые по выбору, не приводят к каким-либо передачам управления, то программа продолжает "идти по префиксам" беспрепятственно.

Обычно оператор, с которым имеет дело переключатель, бывает составным. В начале этого оператора могут быть описания, но инициализация автоматических или регистровых переменных не выполняется.

Оператор `break`.

Оператор `break` приводит к окончанию выполнения ближайшего внешнего оператора `while`, `do`, `for` или `switch`; управление передается оператору, непосредственно следующему за заканчиваемым.

Оператор `continue`.

Оператор `continue` приводит к передаче управления в организующую цикл часть ближайшего внешнего оператора `while`, `do` или `for`, т.е. в конец цикла. Более того, можно сказать, что в каждом из операторов

```
while (...) {      do {      for(...) {
    ...
    contin::;      contin;;    contin;;
}                  } while(...) }
```

оператор `continue` эквивалентен оператору `goto contin`. Следом за `contin` идет пустой оператор.

Оператор `return`.

Выход из функции к точке вызова происходит с помощью оператора `return` одного из следующих двух видов:

```
return;
return <выражение>;
```

В первом случае возвращаемое значение не определено. Во втором случае в точку вызова возвращается значение выражения. Если нужно, то, как и при присваивании, производится преобразование значения выражения

к типу функции, где встречается этот оператор. Выход из функции "через конец" эквивалентен возврату без выдаваемого значения.

Оператор goto.

Безусловную передачу управления можно выполнить с помощью оператора

`goto <идентификатор>;`

Идентификатор должен быть некоторой меткой, помещенной в текущей функции.

Помеченный оператор.

Перед любым оператором можно поставить метку, имеющую вид
<идентификатор>:

Такая конструкция описывает идентификатор как метку. Метка используется только для указания точки перехода в операторе `goto`. Областью действия метки является текущая функция, исключая любые подблоки, в которых этот идентификатор переопределяется.

Пустой оператор.

Пустой оператор имеет вид

Его стоит использовать, если нужно поставить метку перед концом составного оператора или же для задания пустого тела цикла в таких операторах, как `while`.

Оператор asm.

Общий синтаксис:

`asm <код команды> <операнды>`

`<; или <конец строки>>`

Оператор такого вида позволяет вставлять в текст программы коды на языке ассемблера. Все символы Си заменяются на их "ассемблерные" эквиваленты. Воспользоваться этой возможностью из операционного окружения нельзя - необходимо использовать `tcc`.

15. ПРЕПРОЦЕССОР

Компилятор для языка Си имеет препроцессор, который выполняет подстановки для макровывозов, проводит условную компиляцию и вставляет указанные файлы. Командные строки препроцессора начинаются с “решетки” (символа ‘#’) и заканчиваются символом перевода на новую строку. Если непосредственно перед концом строки поставить обратную косую черту (backslash - символ ‘\’), то следующая строка текста будет восприниматься как продолжение предыдущей командной строки.

Синтаксис этих строк не связан с синтаксисом самого языка. Они могут появляться в любом месте программы и влияют на интерпретацию всего нижеследующего текста до конца файла входной программы - обычные правила областей действия на них не распространяются.

15.1. ЗАМЕНА ЛЕКСЕМ

Командная строка имеет вид:

```
#define <идентификатор> <подстановка>
```

Обратите внимание - в конце нет никакой точки с запятой! Такая команда заставляет препроцессор заменять последующие вхождения идентификатора на указанную лексему. Определения такого типа очень хорошо подходят для введения констант - параметров реализации, например, в таких комбинациях:

```
#define TABSIZE 100
int table[TABSIZE];
```

Какого бы вида подстановка ни произошла, результирующая строка вновь просматривается и в ней ищутся определенные идентификаторы. При описании и той, и другой подстановки могут встречаться “длинные” определения, продолжающиеся в следующей строке.

Подставляемый текст может содержать любые неслужебные символы, например, кавычки. Так, после определения

```
#define READ_ERROR    "\nОшибка при чтении файла %s"
можно использовать, скажем, такой оператор:
printf(READ_ERROR, file_name);
```

Если в подставляемом тексте должен встретиться какой-либо служебный символ, используются правила, описанные в параграфе 9.3. Например,

```
#define ENTER    '\n'
```

заставит препроцессор в дальнейшем все встречаемые им лексемы ENTER заменять символьной константой ‘\n’.

Турбо Си не распространяет подстановку внутрь строк (имеются компиляторы, которые это делают). Скажем, если определить ENTER, а после этого вместо приведенного определения READ_ERROR написать

```
#define READ_ERROR "ENTER Ошибка при чтении файла %s"
```

то оператор

```
printf(READ_ERROR, file_name);
```

не переведет строку перед выдачей текста - вместо этого будет выдано сообщение вида: "ENTER Ошибка при чтении файла <имя файла>".

Повторим здесь еще пример, приведенный в параграфе 9.3:

```
#define PROMPT_HEAD "\nНаберите "
```

```
#define PROMPT_TAIL ", затем нажмите Return:"
```

```
static char *decimal_prompt =
```

```
PROMPT_HEAD " десятичное число" PROMPT_TAIL,
```

```
*yes_no_prompt =
```

```
PROMPT_HEAD " \"да\" или \"нет\""" PROMPT_TAIL;
```

В результате работы препроцессора и согласно правилам записи строковых констант начальными значениями переменных `decimal_prompt` и `yes_no_prompt` будут строки

```
"\nНаберите десятичное число, затем нажмите Return:"
```

и

```
"\nНаберите \"да\" или \"нет\", затем нажмите Return:"
```

Команда вида

```
#undef <идентификатор>
```

приводит к тому, что препроцессор начинает считать указанный идентификатор неопределенным, т.е. не подлежащим замене.

15.2. МАКРОСЫ

Если команда имеет вид:

```
#define <ид1>(<ид2>,...) <подстановка>
```

, причем между первым идентификатором и скобкой нет никакого пробела, то она определяет макроподстановку с аргументами. После такого описания вхождение первого идентификатора с последующей открывающей круглой скобкой, лексемами, разделенными запятыми, и закрывающей круглой скобкой заменяется на строку-подстановку из определения. Причем каждое вхождение идентификатора, упомянутого среди формальных параметров определения, заменяется на соответствующую лексему-строку из обращения.

Фактические аргументы обращения суть лексемы-строки, отделенные одна от другой запятой, однако запятая, встретившаяся в кавычках или "защищенная" скобками, уже не считается разделителем для аргументов. Число формальных и число фактических параметров должны быть одинаковыми. На тексты внутри строк или символьные константы в Турбо Си механизмы подстановок не распространяются. Так же, как и после "простой" макроподстановки, результирующая строка здесь тоже вновь просматривается и в ней ищутся определенные идентификаторы.

Директивой

```
#undef <имя макроса>
```

можно заставить препроцессор "забыть" введенное определение.

Пример:

```
#define MEM_ERROR      "Не хватило памяти"

#define memalloc(p,TYPE) ((p)=malloc(sizeof(TYPE)))
if(memalloc(q,float) == NULL) {
    printf(MEM_ERROR);
    exit(-125);
}
```

В результате будет выведено сообщение и программа завершит работу с кодом -125, если функция `malloc()` возвратила `NULL`.

Тот же пример в более изощренном варианте:

```
#define MEM_ERROR      "Не хватило памяти"
#define memalloc(p,TYPE)
    if ( (p)=malloc(sizeof(TYPE)) == NULL) { \
        printf(MEM_ERROR); \
        exit(-125); \
    }
memalloc(q,float)
```

Данная программа будет работать так же, как и предыдущая, но проверяющие, выводющие и выходящие коды не нужно писать после каждого обращения за памятью.

Более изощренный пример макроподстановки:

```
#define myalloc(TYPE) malloc(sizeof(TYPE))
#define mystrdup(str) str == NULL ? NULL : \
    strcpy(myalloc(strlen(str)+1),str)
```

После таких определений оператор вида `p = mystrdup(msg)` будет работать правильно, даже если впоследствии почему-либо окажется выгодным заменить обращение к `malloc()` на вызов собственной функции, управляющей памятью.

Если в макросе соединить две лексемы "решетками" (`##`, между ними пробел не разрешается, а по обе стороны от "решеток" пробелы разрешены), то лексемы будут "склеены", причем без пробелов. Например, вызов `VAR(x,6)` макроса

```
#define VAR(i,j)      (i ## j)
```

даст в тексте программы `x6`. Понятно, что это предпочтительнее, чем нетранспортабельный код вида `(x/**/j)`, и дает больше возможностей.

Если в теле макроса перед одним из аргументов поставить одиночную "решетку", то препроцессор превратит фактический аргумент в строку, окаймленную двойными кавычками. Пусть, например, определение макроса таково:

```
#define TRACE_MSG(code) printf("#code " = "%d\n", code)
```

Тогда фрагмент кода

```
value = 1144;
TRACE_MSG(value);
```

препроцессор превратит в следующий:

```
value = 1144;
printf("value" " = "%d\n".value);
```

если теперь вспомнить о склеивании строк, то мы увидим, что получилось вполне допустимое обращение к `printf()`.

15.3. ВКЛЮЧЕНИЕ ФАЙЛА

Команда вида

```
#include "имя файла"
```

приводит к тому, что вместо этой команды подставляется содержимое всего файла с указанным именем. Поименованный файл сначала разыскивается в директории, где расположен начальный входной файл. Если его там нет, то он ищется в других "стандартных" местах. При другом варианте команды

```
#include <имя файла>
```

поиск идет только в стандартных местах, а не в директории входного файла (см. главу об установке опций).

Во включаемом файле снова могут находиться команды включения.

15.4. УСЛОВНАЯ КОМПИЛЯЦИЯ

Команда компилятора, имеющая вид:

```
#if <константное выражение>
```

проверяет, будет ли отличаться от нуля выражение, составленное из констант.

Команда вида

```
#ifdef <идентификатор>
```

проверяет, определен ли в данный момент в компиляторе указанный идентификатор, т.е. входил ли он в команду вида `#define`.

Команда вида

```
#ifndef <идентификатор>
```

проверяет, является ли в данный момент указанный идентификатор неопределенным.

За любой из этих трех команд может следовать произвольное число строк текста, возможно, содержащих команду

```
#else
```

и заканчивающихся командой

```
#endif
```

Если проверяемое условие справедливо, то строки между `#else` и `#endif` игнорируются - компилируются строки между `#if` и `#else`. Если проверяемое условие не выполняется, то игнорируются все строки между проверкой и командой `#else`, а если ее нет, то командой `#endif`.

Между `#if` и `#endif` может находиться также строка вида

```
#elif <константное выражение>
```

В этом случае проверяется условие, упомянутое в `#if`, и если оно неверно, то строки, находящиеся между `#if` и `#elif`, игнорируются, после чего проверяется условие, упомянутое в `#elif`. Строки, следующие за `#elif`, игнорируются, если и это условие окажется неверным.

Такие команды могут вкладываться одна в другую, но соответствующие друг другу **#if**, **#else**, **#elif** и **#end** должны находиться в одном файле.

Команды **#ifdef** и **#ifndef** могут принимать вид:

```
#if defined(<идентификатор>)
```

и

```
#if !defined(<идентификатор>)
```

причем в выражение после **#if** могут входить **&&** и **||**.

Примеры:

```
#if defined(__TURBOC__) && defined(NEARPOINTERS)
    space = farcoreleft();
#elif defined(FARPOINTERS)
    space = coreleft();
#else
#error Unsupported memory model
#endif
#ifdef DEBUG
    printf("Total space %d\n", space);
#endif
#ifndef Quiet
    PrintDiagnostics();
#endif
```

15.5. УПРАВЛЕНИЕ СТРОКАМИ

Для “удобства” других препроцессоров, которые могут формировать программу на языке Си, команда вида

```
#line <константа> <идентификатор>
```

заставляет компилятор при диагностике “поверить”, что номер следующей строки входного текста такой, который указывает константа, а текущий файл именуется упомянутым идентификатором. Если идентификатор отсутствует, то ранее запомненное имя файла не изменяется.

Пример:

```
#line 55 main.c
```

15.6. ДИРЕКТИВА **#error**

Если в программе встретилась строка

```
#error <сообщение>
```

то компилятор выдаст <сообщение> об ошибке и прекратит работу.

Пример:

```
#if !defined(MODEL)
#error Модель памяти не поддерживается
#endif
```

Сообщение будет выдано, если идентификатор **MODEL** не был определен.

15.7. ДИРЕКТИВА `#pragma`

Турбо Си опознает три вида директив `#pragma`.

Директива

`#pragma inline`

сообщает компилятору, что в программу включен(ы) `asm`-оператор(ы). Лучшее всего помещать эту директиву в самом начале файла, поскольку компилятор перезапускает себя с опцией `-B`, когда встречается такую директиву. Вместе с тем, можно опустить как эту директиву, так и соответствующую опцию компилятора, поскольку компилятор сам перезапустит себя, когда встретит `asm`-оператор. Таким образом, эта директива, как и опция, нужна лишь для того, чтобы сберечь немножко времени работы компилятора.

Директива

`#pragma savereg`

используется в тех случаях, когда при входе в `huge`-функцию нельзя менять содержимое регистров. Эту директиву нужно располагать непосредственно перед определением функции.

Наконец, множество вариантов директивы

`#pragma warn`

используются для того, чтобы изменить опции компиляции, установленные опцией `-wxxx` командной строки или входами меню `Display warnings ...On options`.

Все варианты имеют вид:

`#pragma warn +xxx`

`#pragma warn -xxx`

`#pragma warn .xxx`

`+xxx` включают генерацию соответствующих предупреждений, `-xxx` выключают их. `.xxx` меняют текущее состояние на противоположное.

Приведем список возможных значений `xxx` по группам, как они расположены в меню операционного окружения.

Нарушения правил ANSI (ANSI Violations)

big Шестнадцатеричная или восьмеричная константа слишком велика

Hexadecimal or octal constant too large.

dup Неидентичное перопределение

Redefinition not identical

ret Встречены две формы `return`

Both `return` and `return of a value` used

str Не является частью структуры (относится и к объединениям!)

Not part of structure

stu Неопределенная структура

Undefined structure

sus	Подозрительное преобразования указателя Suspicious pointer conversion	
voi	В void-функции встречен оператор return <значение> functions cannot return a value	Void
zst	Структура нулевой длины Zero length structure	
	Общие ошибки (Common Errors)	
aus	Переменной присвоено значение, которое нигде не использо- вано	
	Assigned a value but never used	
eff	Код никогда не сработает Code has no effect	
par	Неиспользуемый параметр Parameter never used	
pia	Возможно неверное присваивание Possibly incorrect assignment	
rch	Недостижимый участок программы Unreachable code	
rvi	Функция должна возвращать значение Function should return a value	
	Менее общие ошибки (Less Common)	
amb	Непонятный оператор, нужны скобки Ambiguous operators need parentheses	
amp	Применение & к функции или массиву недопустимо Superfluous & with function or array	
nod	Функция не объявлена No declaration for function	
pro	Вызов функции без прототипа Call to function with no prototype	
stv	Структура передана по значению Structure passed by value	
	Предупреждения о транспортабельности (Portability Warnings)	
apt	Нетранспортабельное присваивание значения указателю	
Nonportable	pointer assignment	
cln	Слишком длинная константа Constant is long	
cpt	Нетранспортабельное сравнение указателей Nonportable pointer comparison	
rng	Константа выходит за пределы сравнения Constant out of range in comparison	
rpt	Нетранспортабельное преобразование указателя Nonportable pointer conversion	
sig	Преобразование может вызвать потерю значащих цифр	
Conversion	can lose significant digits	
usc	Смешаны указатели на signed и unsigned символьные перемен- ные	
	Mixing pointers to signed and unsigned char	

15.8. МАКРОИМЕНА, СОДЕРЖАЩИЕ ОБСТАНОВКУ

Препроцессору Турбо Си известны некоторые идентификаторы, которые, следовательно, можно использовать в программе.

__STDC__ Константа равна 1, если для компиляции установлена опция **ANSI keywords only...On**; в противном случае имя не определено.

__DATE__ Дата начала компиляции текущего файла в текстовом виде.

__TIME__ Время начала компиляции текущего файла в текстовом виде.

__FILE__ Имя компилируемого файла - строка.

__LINE__ Десятичный номер компилируемой строки текущего файла.

__TURBOC__ Шестнадцатеричный номер используемой версии Турбо Си. Пример: версия 1.5 представлена в виде **0x0150**.

__PASCAL__ Константа равна 1, если для компиляции установлена опция **"Calling convention...Pascal"**. В противном случае имя не определено.

__CDECL__ Константа равна 1, если для компиляции установлена опция **"Calling convention...C"**. В противном случае имя не определено.

__MSDOS__ Целая константа, равная 1.

Среди нижеследующих имен будет определено одно, соответствующее выбранной модели памяти:

__TINY__ __SMALL__ __MEDIUM__
__COMPACT__ __LARGE__ __HUGE__

ПРИЛОЖЕНИЕ. Синтаксис языка Турбо Си.

```

<описание> :=
    <объявление> |
    <определение функции>
<объявление> ::=
    <список атрибутов -opt>
    <список спецификаций -opt> ;
<список атрибутов> ::=
    <атрибут объявления> <список атрибутов -opt>
<атрибут объявления> ::=
    <атрибут памяти> |
    <атрибут типа> |
    <атрибут функции> |
    cdecl | pascal |
    typedef
<атрибут памяти> ::=
    auto |
    static |
    extern |
    register
<атрибут функции> ::=
    interrupt | near | far
<атрибут типа> ::=
    <модификатор переменной> |
    void |
    <спецификатор арифметического типа> |
    <спецификатор составного типа> |
    enum |
    <typedef-имя> |
    <enum-атрибут>
<модификатор переменной> ::=
    const | volatile
<спецификатор арифметического типа> ::=
    <модификатор арифметического типа>
    <спецификатор арифметического типа -opt> |
    char | int | float | double
<модификатор арифметического типа> ::=
    short | long | signed | unsigned
<спецификатор составного типа> ::=
    struct | union

```

```

<список спецификаций> ::=
    <спецификация> |
    <спецификация> , <список спецификаций>
<спецификация> ::=
    <спецификация имени> <инициатор -opt>
<спецификация имени> ::=
    <идентификатор> |
    (<спецификация>) |
    <спецификация указателя> |
    <спецификация массива> |
    <спецификация составного типа> |
    <спецификация перечислимого типа> |
    <спецификация функции>
<спецификация указателя> ::=
    <модификатор указателя -opt>
    * <модификатор переменной -opt>
    <идентификатор>
<модификатор указателя> ::=
    near | far | huge | _es | _ss | _cs | _ds
<спецификация массива> ::=
    <модификатор указателя -opt>
    <идентификатор> [ <константное выражение> ]
<спецификация составного типа> ::=
    <идентификатор -opt> { <список полей> }
<список полей> ::=
    <спецификация поля> ; <список полей -opt>
<спецификация поля> ::=
    <атрибут типа> <спецификация> |
    <спецификатор арифметического типа -opt>
    <идентификатор -opt> :
    <константное выражение>
<спецификация перечислимого типа> ::=
    { <список значений> }
<список значений> ::=
    <имя с инициатором> |
    <имя с инициатором> , <список значений>
<имя с инициатором> ::=
    <идентификатор> |
    <идентификатор> = <целая константа>
<спецификация функции> ::=
    <декларация функции> |
    <прототип функции>
<декларация функции> ::=
    <идентификатор> (
<прототип функции> ::=
    <идентификатор> ( void ) |
    <идентификатор> ( <список аргументов> )
<список аргументов> ::=
    ...
    |

```

```

<тип аргумента> |
<тип аргумента> , <список аргументов> |
<объявление> |
<объявление> , <список аргументов>
<тип аргумента> ::=
    <список атрибутов типа>
<список атрибутов типа> ::=
    <атрибут типа> <список атрибутов типа -opt>
<инициатор> ::=
    = <выражение> |
    = { <список инициаторов> } |
    = { <список инициаторов> , }
<список инициаторов> ::=
    <список инициаторов> |
    <список инициаторов> , <список инициаторов> |
    { <список инициаторов> }
<определение функции> ::=
    <заголовок функции>
        <список объявлений аргументов -opt>
        { <тело функции> }
<заголовок функции> ::=
    <идентификатор> ( void ) |
    <идентификатор> ( <список имен аргументов> ) |
    <идентификатор>
        ( <список формальных параметров> )
<список имен аргументов> ::=
    ... |
    <имя> |
    <имя> , <список имен аргументов>
<список объявлений аргументов> ::=
    <объявление> ;
    <список объявлений аргументов -opt>
<список формальных параметров> ::=
    ... |
    <объявление> |
    <объявление> , <список формальных параметров>

```

ОГЛАВЛЕНИЕ

Предисловие издателя.....	3
Предисловие.....	5
Часть I. Операционное окружение Турбо Си.	
1. Основные возможности.....	7
1.1. Окна.....	7
1.2. Меню.....	9
1.3. Система помощи.....	16
2. Редактор.....	18
2.1. Характеристика редактора.....	18
2.2. Режимы редактирования.....	19
3. Компилятор.....	25
3.1. Как получить выполняемую программу.....	25
3.2. Команды, запускающие работу над проектом.....	27
4. Сборка выполняемой программы.....	47
4.1. Линкер и его "обязанности".....	47
4.2. Опции линкера.....	48
5. Управление проектом.....	53
5.1. Многомодульные программы.....	53
5.2. Файл проекта.....	57
5.3. Меню управления проектом.....	59
6. Отладка в интегрированной среде.....	62
7. Работа из командной строки.....	71
8. Утилиты.....	85
8.1. BGIOBJ.....	85
8.2. CINSTXFR.....	86
8.3. CPP.....	86
8.4. GREP.....	87
8.5. OBJXREF.....	90
8.6. TCCONFIG.....	92
8.7. TCINST.....	93
8.8. THELP.....	97
8.9. TLIB.....	100
8.10. TOUCH.....	103
Часть II. Справочное руководство по Турбо Си.	
9. Лексика.....	105
9.1. Комментарии.....	105
9.2. Идентификаторы.....	106
9.3. Константы.....	106
9.4. Зарезервированные слова.....	110
10. Описания.....	112
10.1. Имена.....	112
10.2. Нотация для синтаксиса.....	112
10.3. Состав описаний.....	113
10.4. Атрибуты объявлений.....	113
10.5. Спецификации в объявлении.....	118
10.6. Инициализация.....	121
10.7. Смысл объявлений.....	122
10.8. Названия типов.....	124
10.9. Определение функции.....	124
11. Описания переменных.....	126
11.1. Объекты и их имена.....	126
11.2. Объекты исходных типов.....	127
11.3. Ссылочные типы и массивы.....	128
11.4. Перечислимые типы.....	131
11.5. Структуры и объединения.....	133
11.6. Модификаторы const и volatile.....	137

12. Описания функций.....	139
12.1. Основные сведения о функциях.....	139
12.2. Аргументы функций.....	143
12.3. Функции с переменным числом аргументов.....	146
12.4. Указатели.....	147
12.5. Функция <code>main()</code>	149
13. Операции и выражения.....	152
13.1. Предварительные замечания.....	152
13.2. Арифметические операции.....	153
13.3. Побитовые операции.....	155
13.4. Логические операции.....	156
13.5. Операции сравнения.....	157
13.6. Операции присваивания.....	158
13.7. Операция "запятая".....	159
13.8. Адресные операции.....	160
13.9. Операция условия.....	161
13.10. Операции над массивами.....	161
13.11. Операции над структурами и объединениями.....	161
13.12. Другие операции.....	162
13.13. Старшинство операций.....	162
13.14. Преобразования.....	163
13.15. Константные выражения.....	165
14. Операторы.....	167
15. Препроцессор.....	171
15.1. Замена лексем.....	171
15.2. Макросы.....	172
15.3. Включение файла.....	174
15.4. Условная компиляция.....	174
15.5. Управление строками.....	175
15.6. Директива <code>#error</code>	175
15.7. Директива <code>#pragma</code>	176
15.8. Макроимена, содержащие обстановку.....	178
Приложение. Синтаксис языка Турбо Си.....	179
Оглавление.....	182

УЧЕБНО-НАУЧНЫЙ ЦЕНТР

“ТРЭК”

ПРЕДЛАГАЕТ СОТРУДНИЧЕСТВО

обладателям авторских прав:

- в распространении оригинальных программных и технических разработок;
- в реализации интересных идей;
- в издании печатной продукции по вычислительной технике, менеджменту, по другой тематике.

Наш адрес:

107078, Москва, ул. Новая Басманная, д. 16 а, УНЦ “ТРЭК”.

Контактный телефон: 263-94-76.

