В. Г. АБАШИН

СИНЕРГИЯ СИ

Москва 2019

УДК ____.__:___.__(___._)
ББК __.___._-___._
____

Абашин В.Г.
A13 Синергия Си: Учеб. пособие. - М: XXXXXXX
Издательство, 2019. - 520 с.: ил.
ISBN _-___-_____-_.

Первые две главы содержат общие знания, необходимые для освоения языка программирования Си. Главы третья, четвертая и пятая содержат описание языка Си. В них приводятся ссылки на разделы стандарта языка Си, содержащие исчерпывающие описания конструкций языка Си. Шестая глава посвящена наиболее часто используемым технологиям программирования в среде GNU/Линукс. В седьмой главе затронуты вопросы алгоритмизации. Восьмая глава состоит из списка этапов разработки программы на языке Си с учетом технологий программирования.

Для желающих освоить программирование на языке Си.

УДК ___._._.__:___.__(___._)
ББК __.___._-___._

ISBN _-___-_____-_.

Посвящаю своему сыну Абашину Андрею Валерьевичу

Выражаю благодарность моим учителям:

- Сорокиной Галине Александровне,
- Брагиной Людмиле Николаевне,
- Лобановой Валентине Андреевне,
- Пилипенко Ольге Васильевне,
- и всем кто смог меня научить чему-то, независимо от обстоятельств.

Синергия Си

Введение

Тест на то, какой Вы программист.
Что Вы видите?

```
i = 0;
```

Варианты ответов:
а) код;
б) выражение с ошибкой;
в) так уже не делают;
г) все равно на это книгой не мотивируешь.

Список с оценкой квалификации и еще один шуточный тест где-то во введении.

Зачем становиться программистом?

Заработная плата программиста обычно больше, чем у других специалистов. Это элита любой компании. Отличный старт для управленцев и менеджеров. Это возможность посмотреть мир, имея высокий личный доход, видеть перспективные технологии до их появления в серийном производстве, а значить жить немного в будущем.

Основная цель среды «информационщиков» - делать мир лучше. Возможно, что такой подход немного идеализирован, это скорее то, к чему стремятся все программисты. Все равно, любое свое значимое решение следует оценивать в соответствии с этой миссией. Получившие широкое распространение компьютерные злоумышленники обычно имеют посредственную квалификацию или не имеют специального образования вообще. Это факт, это статистика правохранительных органов.

Само по себе программирование похоже на плавание. В том смысле, что чтобы научиться плавать —

необходимо плавать. По учебнику плавать не научишься. Учебник должен лежать рядом с клавиатурой, с помощью которой проверяется все, что написано в учебнике. Программирование — это плавание для ума.

С другой стороны, программирование похоже на игру. Каждый день или даже каждый час необходимо ставить себе задачу, решать головоломки и получать небольшую порцию серотонина в награду за её успешное решение. Программирование позволяет развивать свой интеллект, изучать полезные в обычной жизни приемы мышления. В то же время программирование требует усидчивости.

Кроме желания заниматься программированием, для успешного освоения этой профессии необходимо обладать рядом качеств. Наиболее яркой профессиональной деформацией среди программистов является тип мышления. Каждый современный программист обычно обладает несколькими типами мышления. Самый важный — **алгоритмический тип мышления**. Как показывает практика на его формирование уходит не менее полугода. Время его развития зависит от исходных условий. Человек, обладающий математической подготовкой, обычно уже имеет алгоритмическое мышление и требуется только тренировка в переложении алгоритмов на язык программирования или отображение алгоритмов с помощью блок-схем. Охарактеризовать данный тип одним предложением можно так: **все — последовательность действий**.

Другой популярный тип мышления – **объектный**. Он реализуется объектно-ориентированными языками. Его развитие обычно происходит после формирования алгоритмического мышления и занимает немного меньше времени. Вполне достаточно месяца, чтобы

разобраться с этой концепцией. Самым близким объектно-ориентированным языком к Си является C++, однако это полностью другой язык. Объектно-ориентированное мышление можно реализовывать и с помощью языка Си, но такой подход может потребовать излишних или «некрасивых» конструкций в коде, которые будут признаны Си — программистами плохими практиками программирования. Охарактеризуем этот тип мышления так: **все — объект, объекты взаимодействуют между собой**.

Еще один популярный тип мышления — **многопоточный**. Сейчас его не выделяют как отдельный тип мышления, считая скорее рядом ограничений, зависимых от современных технологий, например **CUDA**. Данный тип мышления является надстройкой как для алгоритмического, так и для объектного типа мышления. Обучение ему обычно происходит на месячных курсах повышения квалификации для программистов с опытом или в рамках дисциплины в среднем или высшем учебном заведении. Охарактеризуем этот тип мышления так: **вычисления – это потоки, потоки текут с разной скоростью, но должны закончится вместе**.

Главным ошибочным типом мышления является – **предоптимизация**. Он встречается как у начинающих программистов со стажем 1-3 года, так и у продолжающих, со стажем 3-6 лет. В этом случае программист пытается улучшить программу до того, как она написана полностью. Этот тип мышления — главный враг программистов. Любая оптимизация заключается в объединении ресурсов или алгоритмов. Получается своего рода свернутая, упакованная структура программы, которая, в случае изменения, часто требует полного перепроектирования части программы. Если

таких участков в одной программе множество, программу приходится разрабатывать заново.

Для каждого программиста характерен свой способ написания исходных кодов программ. Такой способ называется **стилем программирования**. В данном случае подразумевается, что язык программирования позволяет написать одно и то же действие разными способами, кроме того, можно по разному поставить скобки, сделать для наглядности переносы на новую строку или разрядить код пробельными символами, называть по определенным правилам пользовательские переменные и функции. Обычно в компаниях имеется некоторый набор примеров кода, которые используются для образца. Нарушать корпоративный стиль нельзя. Это не является обязательным навыком, поэтому программисты, получившие образование в постсоветской школе, часто используют ничего не значащие названия переменных из одной буквы и прочие приемы, понижающие читаемость исходных кодов программы.

Любая сложная технология есть плод труда множества людей разных профессий. Технология может рождаться столетиями, а может за один год. На сегодняшний день нет такого человека, который один владел бы всеми технологиями производства вычислительных машин. Не существует даже такого государства, которое одно могло бы произвести самое передовое устройство, без приобретения материалов, технологий или составных элементов в других странах. Каждый элемент информационных систем был в свое время новшеством, которое сделало использование вычислительных средств немного проще. Все это было бы невозможно без формирования среды межчеловеческого общения с позитивным настроем, желанием бескорыстно помогать друг другу в развитии технологий. Такой подход

реализован в различных форумах, конференциях, сообществах, группах, в которых происходит обсуждение развития технологий, обучение молодых специалистов. Базовым законом таких сообществ является желание бескорыстно помогать другим и возможность, в случае необходимости, получить помощь самому.

В сфере информационных технологий при общении принято ссылаться на первоисточник знания о технологии. К ним относятся стандарты и спецификации, принятые в разных странах и организациях: ГОСТ, RFC, IEEE, документы SIG по направлениям. Причина в том, что информационные технологии развиваются по принципам инженерного дела. К программистам на языке Си это относится даже больше, чем, например, к программистам на PHP или других высокоуровневых прикладных языках.

Информационные системы — первая область деятельности человека, в которой удалось создать технологию, используемую большей частью населения планеты, без принуждения одного человека другим. При этом продукт получился более высокого качества, чем при использовании обычных иерархическо-подчиненных структур управления коммерческих организаций. В данном случае имеется ввиду операционная система Линукс, над разработкой которой работают сотни тысяч людей, не имея единого центра управления. Этот факт еще требует осознания обществом, но несомненно он окажет значительное влияние на развитие всей человеческой цивилизации.

Ответьте себе на следующие вопросы. Вам хотелось бы научиться пользоваться инструментом, который является ключом ко всем современным информационным технологиям? Какой язык программирования является самым универсальным? На каком языке можно писать

программы для микроконтроллеров по цене 135 рублей, программировать Интернет-вещи, создавать искусственный интеллект? Какой язык является одним из самых компактных и простых?

Всем этим условиям удовлетворяет язык программирования Си. Этому языку программирования 46 лет. Программисты на этом языке руководствуются принципом: **чем проще — тем лучше**. Наиболее продвинутые используют специальные правила даже для обоснования применения вполне тривиальных конструкций, советуют избегать longjump указателей на функции, malloc, арифметики указателей, обычных структур.

Следует учитывать, что когда работодатель формулирует вакансию программиста на языке Си, он не ищет человека, который умеет программировать на СИ. Работодатель ищет человека, который умеет решать задачи, стоящие перед компанией. Обычно это программирование микроконтроллеров, высокоскоростные вычисления, большие данные, робототехника. Соискатель должен не просто знать как записываются операторы, а иметь несколько видов мышления, алгоритмическое и объектно-ориентированное, знать технологию программирования, иметь теоретическую подготовку в предметной области, быть активным участником сообщества разработчиков и знать, как устроены технологии, эксплуатируемые на текущий момент в разных компаниях.

Только после овладения рядом навыков происходит прорыв возможностей разработчика, у него появляется возможность создавать краткие, эффективные решения. Этот потенциал приводит к синергетическому эффекту, являющемуся симбиозом техники и культуры, **синергии Си**.

В качестве информации для разработки профессиональные программисты используют технологические стандарты или документацию, поставляемую компанией разработчиком. Все остальные источники считаются уделом начинающих. Фактически, полностью читать все стандарты на все используемые технологии не приходится. Обычно достаточно ограничиться рядом разделов, описывающих необходимые функциональные возможности.

Сами стандарты обычно тяжело читаются и описывают то, как написать ту или иную конструкцию, но учиться программированию по ним чрезвычайно сложно. В стандарте часто используются упреждающие ссылки на материал, который будет изложен позже, материал может излагаться по частям, в соответствии с разделом. Текущей версией стандарта языка программирования Си является версия от 2018 года. Документ, являющийся стандартом языка Си, стоит дорого, однако бесплатно доступен черновик предыдущей версии стандарта с номером документа ISO/IEC9899:2017. Он содержит неточности, которые не мешают использовать его для работы. Для его поиска необходимо произнести номер документа любому программному ассистенту в своем мобильном устройстве.

Стандарт языка программирования появился как развитие книги Брайна Кернигана и Дениса Ритчи «Язык программирования Си», называемой в среде разработчиков K&R. Сама книга раскрывает красоту языка и его парадигму, однако не уделяет внимания современным технологиям создания программ. Их тогда просто не существовало.

Современные учебники по программированию на языке Си, используемые в учебных заведениях, больше сосредоточены на развитие компетенций, требуемых в

соответствии с учебным планом. Они эффективно обучают решать задачи в рамках курса и редко выходят за его рамки.

Не стоит игнорировать еще один эффективный источник знаний по программированию — форумы и прочие способы общения в сети Интернет. Единственный совет — при использовании Интернета соблюдать правила общения, характерные для конкретной группы. Очень рекомендуется **не использовать сленг**. Он очень быстро меняется и следует моде. Наиболее квалифицированные ответы пишутся грамотным языком носителя, например, на русском или английском. Кроме того, в сети отвечают тем, кто описывает конкретную проблему. Например, на вопрос «я хочу и не понимаю как, напсать программу для работы с папками», ответа скорее всего не последует. Появятся только отписки и просьбы уточнить. Правильно заданный вопрос должен звучать примерно так: «функции из каких заголовочных файлов используются для отображения списка каталогов». Таким образом производится уточнение задачи, а также демонстрируется, что работа уже начата и ведется, требуется немного подсказать, чтобы программа была дописана. Для разработчика на языке программирования Си двумя основными интернет ресурсами являются cplusplus.com и stackoverflow.com.

Если Вам интересно программирование и Вы дочитали до этой страницы, представим задачи, которые ставились перед этой книгой. Она должна:

1. Заставить обратить внимание на стандарт языка программирования, получить сведения об его структуре и научить работать с ним.

2. Научить читать исходный текст программы в слух, а также вспомнить о том, что программы можно писать без компилятора. Такой подход единственно возможный,

когда сборка проекта занимает около суток.

3. Сразу учиться программировать правильно. Обычно авторы учебников жалеют своих читателей, в результате читателям таких книг приходится переучиваться в процессе программирования.

4. Обратить внимание начинающих программистов на умение писать код программ на бумаге без компьютера, что значительно снижает количество логических ошибок в программе, но повышает количество синтаксических ошибок, от которых избавляются, используя компилятор.

Для работы с этой книгой желательно иметь под рукой черновик стандарта языка программирования 2017 года и K&R. «Синергия Си» несет некоторую нелогичность повествования. Описание переменных произведено в одном разделе, способ их задания в разных системах счисления в другом, указатели для работы с ними в третьем. Также встречается их повторное описание в случае необходимости.

Другие нарушения, умышленно допущенные в книге «Синергия Си»:

- автор не придерживается одного стиля программирования, чтобы читатель не привыкал;
- один и тот же алгоритм может быть реализован по разному в разных примерах;
- наличие повторяющихся объяснений одинаковых участков исходного текста, которые позволяют читать примеры без перечитывания предыдущих глав;
- в нескольких случаях дается не только текст программы, но и пояснения, как его улучшить.
- избыточно подробные объяснения для простых выражений текста программы.
- не рассматриваются технологии совместной разработки, за исключением git.

Избыточное описание связано с тем, что неизвестно, в каком месте у программиста может возникнут трудность. Кроме того, к книгам иногда обращаются после появления ошибок, не поддающихся быстрой отладке. В этом случае чтение подробного описания помогает посмотреть на проблему с другой стороны, сдвинуться с места и решить проблему.

Первые две главы содержат информацию, которая мгновенно всплывает в памяти у автора книги, когда он начинает программировать. Если читатель не является студентом специализированного высшего учебного заведения, обычно эти темы остаются не изучены.

Книга адресована в первую очередь тем, кто хочет начать изучение языка программирования Си. Кроме того, она будет полезна тем, кто хочет сменить используемую при разработке программ операционную систему на Линукс.

Желательно иметь опыт работы с компьютером не ниже продвинутого пользователя. Однако если этого уровня нет, не стоит отчаиваться. Разве это может стать преградой для достижения Вами Вашей цели?

Кстати, результаты тестирования проведенного в начале введения:

а) начинающий;

б) программист;

в) хороший программист;

г) эксперт.

В конце еще один шуточный тест. Как определить стаж программиста:

- 1-5 года – пишет как получится, недооценивает сроки разработки, легко заводит разработку в тупик;

- 5-10 лет – делает все правильно, … в теории;

- 10-15 лет – не смешивает этапы проектирования и разработки, отталкивается от задачи и ресурсов, сроки

разработки близки к реальным;

- 15+ - решает вашу проблему.

Вы без всяких тестов можете понять, что стали хорошим программистом, когда Вам станет понятно, что эта книга про людей, которые живут технологиями, тратят свое время на их развитие.

Нужна ли именно эта книга? Посмотрите главу 8. Если все, что там написано, Вам известно, эта книга Вам скорее всего не пригодится.

1. Базовые понятия и навыки
1.1. Информация и данные

Первой задачей, подтолкнувшей развитие вычислительных машин, была задача управления со скоростями реакции значительно выше доступной для человека. Известно, что комфортной по физиологическим соображениям, скорость реакции равна примерно 0,2 секунды. Это целых 200 млс. Минимальная, в смысле наилучшая, скорость реакции может ровняться 30 млс и даже быстрее, но работая с такой скоростью, высока вероятность ошибки, и чем дольше длится работа в таком режиме, тем больше ошибок совершает человек.

Во времена зарождения ракетной техники, которая, как и любая другая технология, сначала служила исключительно военным целям, не существовало точных карт в континентальном масштабе. Расчет размера материка по картам того времени приводит к появлению ошибки в десятки километров. Корректировка ошибки требовала оперативного управление ракетой в промежутки времени, на порядки меньшие доступных реакции человека.

Исследуя проблемы управления, ученые пришли к выводу, что они едины для всех сущностей нашего мира, включая искусственные, биологические и социальные системы. Первым заметным трудом в этой области стала книга Норберта Винера «Кибернетика или управление и связь в животном и машине», выпущенная в 1948 году. Суть управления можно описать как использование малой силы для управления большой силой. На сегодняшний момент известно всего четыре вида управления, которые изучает наука «Теория управления».

В основе управления лежит базовая сущность нашего мира – информация. **Информация** не определяемое понятие, также как понятие «точка» в математике. В отличие от **вещества** и **энергии** мы не можем воспринимать информацию органами чувств непосредственно. Единственным органом, взаимодействующим с ней, является мозг человека. Опираясь на это базовое понятие, мы можем строить производные понятия, используемые в нашей деятельности.

Все, что мы воспринимаем в окружающем мире, мы воспринимаем через **отличие** одной сущности от другой. Например, когда мы говорим «ключ от двери», мы обозначаем только применение предмета относительно дверного замка, но не раскрываем сути этого предмета. Любое другое обозначение ключа от двери также будет относительным, относительно другой сущности или действия. Например, ключ металлический, т.е. подразумевает наличие предметов из других материалов, но не говорит о сути дверного ключа. Отличия любой сущности от другой называются **свойствами**. Основным отличием информации от других базовых сущностей этого мира является её неистощимость и подразумевает другие методы получения, создания и преобразования. При этом информация является ресурсом, который относится к любой сущности нашего мира. Наиболее корректно описывать свойства информации через действия, производимые с информацией. Наиболее часто встречающимися из них являются:

Запоминаемость – возможность хранить информацию, используя как среду, так и процессы этой среды. Похожий процесс существует у материи и энергии. Например, можно положить в карман ключи или сохранить энергию в энергоемком материал батареи.

Передаваемость – возможность передачи информации через каналы связи, в том числе с помехами. При этом информация может быть сохранена как в источнике, так и в получателе, а пространственные отношения между частями системы, передающими между собой информацию, не являются существенным ограничением. Это свойство было подробно изучено Клодом Шенноном, результат работы закреплен в виде теорем. Сравнивая с материей и энергией, приведем пример водопровода и передачи энергии в электрической сети. Однако это сравнение не совсем корректно, так как передача материи и энергии зависимо от пространственной распределенности.

Воспроизводимость информации подразумевает возможность создавать копии информации, не изменяя её источник. Получается, информация является неиссякаемым источником и остается равной самой себе при любом количестве копирований. Аналогов свойства воспроизводимости информации для материи и энергии не существует.

Преобразуемость – информация может изменять форму своего существования, однако есть ограничение, связанное с тем, что при преобразовании информации меняется её форма представления, но возрастать она не может. Для материи примером этого свойства служит растворение кристаллов в воде и восстановление их в процессе выпаривания. Энергия может преобразовываться, например, из механической в тепловую или электрическую и обратно.

В процессе преобразования информация может уменьшаться и частным случаем преобразования является **стираемость** информации. В этом случае её количество уменьшается и становится равным 0. На сегодняшнем этапе развития человечеству известен

способ преобразования энергии в материю и обратно через ядерные реакции, однако в естественной среде они нам недоступны.

Таким образом, информация обладает свойствами, аналогичными свойствам энергии и материи, совпадающими с ними с ограничениями и отличными от них.

Информация окружает нас всегда и везде. На сегодняшний день мы научились использовать только наиболее важную часть информации, которая записывается и обрабатывается. Информация, полезная для решения наших задач, называется данными.

Данные – это зарегистрированные сигналы, которые являются представлением информации в заранее определенном, в смысловом понимании, виде и пригодном для дальнейшего использования. Данные упаковываются в сообщения.

Сообщение - это заранее определенная форма и представление передаваемой информации. Например, текст на бумаге имеет определенную форму в виде бумаги и представление в виде букв.

Сообщение подразумевает наличие «**источника»**, «**получателя»** и «**канала связи»**. Смысловые значения «источника» и «получателя» не требуют расшифровки, под «каналом связи» понимается любая среда, с помощью которой сообщение передается от источника получателю.

Основными составляющими «канала связи» являются: «**носитель информации»**, «**сигнал»** и «**параметр сигнала»**.

Под «носителем информации» подразумевается некоторая материя.

«Сигнал» - это сообщение, но не в общем виде, а передаваемое с помощью носителя информации. Обычно

под сигналом подразумевают изменяющийся во времени физический процесс. Например, изменение напряжения в течение секунды.

«Параметр сигнала» - это характеристика, используемая для предоставления сообщения. Например, изменения напряжения от 0,2 до 1,4 В.

Распространенным заблуждением, связанным с неточным переводом иностранной литературы, стало использование понятия «форма представления информации», применительно к понятиям текстовая, графическая, мультимедийная информация. Обычно в этом случае подразумевают формат представления файла, например, mp3 для звукового файла или txt для текстового файла.

На вопрос формы представления информации убедительного ответа пока нет, но в соответствии с теорией информации существует всего две формы представления сигналов информации: **дискретная** и **аналоговая (непрерывная)**. Большинство окружающих нас бытовых вычислительных устройств обрабатывают информацию в дискретном виде. Наиболее часто это **двоичная** информация, но она является частным случаем дискретной информации.

Дискретным сигнал считается в том случае, если его параметр принимает последовательное во времени конечное число значений. Сообщения, передаваемые с помощью дискретного сигнала, называются дискретными, а информация – дискретной информацией.

Если источник вырабатывает непрерывное сообщение с заранее неизвестным количеством состояний, то это соответственно непрерывное сообщение, а информация – непрерывная информация. Рассмотрим пример: человек с помощью речи передает информацию другому человеку. Речь является

моделированной звуковой волной, в которой параметром сигнала является давление, создаваемое в ухе человека. Такие абстрактные примеры очень важны, так как современный уровень информационных технологий давно шагнул за рамки компьютерных вычислений и позволяет взаимодействовать с окружающей средой.

В окружающем нас мире мы чаще встречаемся с непрерывной информацией, однако компьютеры обычно обрабатывают дискретную информацию. В связи с этим востребовано преобразование сигналов от аналоговой в дискретную форму и обратно. Процесс преобразования информации из аналоговой формы в дискретную называется **дискретизацией**. Следует учесть, что обычно дискретизация выполняется с потерей части информации.

Сообщение «на улице пошел дождь» содержит разное количество информации для человека, собирающегося на улицу, и человека, пришедшего домой и не планирующего выходить из дома. В связи с этим возникает вопрос о количестве информации в сообщениях и возможности количественного измерения информации.

Существует два подхода к измерению количества информации. Первый, называемый **вероятностным**, разработан Клодом Шенноном. Он состоит в измерении количества информации относительно её отклонения от полной **энтропии**. Под энтропией, в этом случае, понимается неопределенность. Переформулируем. Объем информации — это степень отклонения от состояния полного хаоса. На формирование данного подхода основное влияние оказали теории физических процессов, в первую очередь законы термодинамики. Вероятностный подход используется чаще при создании информационных систем, взаимодействующих с

аналоговой информацией, иногда для выполнения преобразования информации со смысловым содержанием, например, поиске зависимости в большом объеме данных. Основным ограничением вероятностного подхода является игнорирование смысла и ценности сообщений. Исходя из ограничений вероятностного подхода, для его использования со смысловой информацией требуется применение дополнительных упрощений в расчетах в каждом конкретном случае.

С математической точки зрения вероятностный подход описывается формулой **Хартли** и формулой **Шеннона**. Формула Хартли описывает объем информации для дискретных сообщений с известным основанием алфавита, когда события происходят равновероятностно. Формула Шенона позволяет вычислять объем информации для событий с разной степенью вероятности.

Из формулы Хартли следует, что один бит — это результат события с двумя исходами, например, результатом может быть только да или нет, причем и да и нет выпадают одинаковое количество раз.

Второй подход называется **объемным**, использует понятие бит и обусловлен необходимостью экономии физических ресурсов. С ним мы встречаемся каждый день, например, когда оцениваем размер видеозаписи и запрашиваем размер доступной памяти своих устройств. Во всех устройствах размер памяти указывается в смысле объемного подхода и подразумевает, что в каждой ячейке памяти может оказаться значение 0 или 1 с одинаковой вероятностью. На сегодняшний день развитие технологий позволяет записывать в одну ячейку памяти более 1 бита, однако это не изменяет смысла объемного подхода. Если ячейка позволяет записывать более 1 бита, она воспринимается как некоторое

устройство, осуществляющее групповую запись и чтение бит. Однако производители вычислительных устройств отдают предпочтение двоичным устройствам по причине их простоты и дешевизны производства.

Как следует из описания подходов к измерению информации, ограничивающим является объемный подход. Нельзя записать больше информации, чем может хранить физическая среда. Однако, следуя вероятностному подходу, не вся информация, записанная в память вычислительного устройства, полезна.

Вычислительные устройства обрабатывают данные в цифровом виде и умеют выполнять только логические и математические операции. Начинают появляться устройства, производящие матричные и параллельные вычисления (нейропроцессоры, оптоэлектроника и прочие), однако их базовые функции остаются прежними.

Принятый способ записи чисел и сопоставление им реальных значений называется **системой счисления**, которые делятся на **позиционные** и **непозиционные**. Пример позиционной системы счисления — арабские цифры, непозиционной — римские. Как известно, значение числа в римской системе записи цифр не влияет на её значение, в арабской влияет. В зависимости от позиции цифра может означать количество единиц, десятков и т.д.

Число знаков в позиционной системе счисления называется **основание системы счисления**. В вычислительной технике наиболее часто встречаются следующие системы счисления: двоичная, восьмеричная, десятеричная, шестнадцатеричная.

Преобразование числа из одной системы счисления в другую выполняется по соответствующим формулам или с помощью специальных приемов, подробно описанных в соответствующих источниках.

Изложенной выше информации достаточно для пояснения основ кодирования информации в вычислительных машинах.

Сообщения передаются с помощью **букв**, которые являются элементом некоторого конечного множества отличных друг от друга знаков. Множество знаков с определенным порядком называются алфавитом.

1.2. Кодирование информации

Впервые человечеству кодирование информации потребовалось при разработке систем передачи информации. Проблемы, возникающие при передаче информации, известны со времен, когда для сообщения о нападении использовались сигнальные костры. Хворост и дрова могли намокнуть, костер мог быть разведен другими людьми, заставу, ответственную за разведение костра, могли разгромить. Все эти факты условно можно считать помехами, возникающими в каналах связи.

Одним из пионеров, исследовавших проблему передачи информации в каналах связи с научной точки зрения, был Клод Шеннон, который описал основные проблемы, возникающие при передаче информации, и сформулировал теоремы, позволяющие их решить. К основным проблемам передачи информации, искажаемой помехами, относятся: установления факта искажения информации, выяснение места искажения информации, исправление ошибки с некоторой долей достоверности.

В естественном языке, для защиты от помех, люди используют **избыточность** передаваемой информации. Избыточность является основным средством борьбы с помехами и присутствует в подавляющем большинстве технических систем. Наиболее простым примером избыточности является многократная передача одной и той же информации, до тех пор, пока она не будет принята приемником информации без искажения. С технической точки зрения, избыточность приводит к перерасходу ресурсов, в первую очередь, времени затрачиваемого на передачу информации и объема хранимой и передаваемой информации.

В дальнейшем будет показано построение оптимальных информационных систем, не содержащих

избыточность, это отдельная задача максимальной оптимизации, которая на практике решается крайне редко. Кроме того, оптимальная информационная система все равно будет содержать избыточность в ряде своих компонент, например, в алгоритмах микропрограмм аппаратного обеспечения.

Вывод оптимального объема избыточности основывается на **двух теоремах Шенона**, упрощенно смысл которых сводится к следующему:

- **Первая теорема** дает математический аппарат для оценки минимально допустимого количества двоичных символов, представляющих другой алфавит. Например, если алфавит содержит 5 символов, то будет использоваться минимум 4 двоичных символа, позволяющих кодировать 8 символов. Соответственно 8-5=3, 3 символа двоичной кодировки никогда не будут использоваться.

- **Вторая теорема** декларирует возможность всегда найти такую систему, которая при наличии помех передаст информацию с заданной степенью достоверности. Также она накладывает следующие ограничения: производительность канала всегда должна быть выше производительности источника сообщений, так как кроме информации необходимо передавать избыточные данные.

Считается, что в канале связи сообщение может преобразоваться из символов одного алфавита в символы другого алфавита. **Кодом** называется **правило** такого преобразования, изменяющее вид сообщения. На бытовом уровне понятие кода часто путается с понятием шифрование, которое относится в другой области знания и связана с сокрытием информации, а не изменением его вида. **Применение кода** к некоторому сообщению называется **кодированием**. Обычно под кодированием

подразумевают преобразование информации между источником информации и каналом с помехами, а **декодированием** — преобразование между каналом с помехами и приемником информации. В свою очередь, **кодировщик** и **декодировщик** — это устройства, выполняющие кодирование и декодирование.

Наиболее часто встречающаяся задача, связанная с кодированием, это преобразование текста в различных форматах. Сегодня популярны кодировки ASCII – для аппаратного обеспечения и UTF-8 в сети Интернет. Для русского алфавита в ASCII выделяют кодировки расширенной части таблицы KOI8-R, WIN-1251. Общепринятые таблицы символов описаны в соответствующих стандартах, для UTF-8 это RFC 3629 и ISO/IEC 10646, для KOI8-R это RFC 1489. Также существует ряд ГОСТ (государственных отраслевых стандартов), описывающих различные кодировки.

При изучении любой технологии особое внимание следует уделить поиску технологических стандартов на изучаемую технологию, таких как ГОСТ, RFC, ISO. В мире высококвалифицированных программистов и инженеров знание о существовании стандартов считается обязательным. Обычно так проверяется квалификация собеседника. Если вопрос звучит как просьба о помощи в поиске стандарта на конкретную технологию или уточнение по какому-либо пункту этого стандарта, практически всегда общение будет строиться в позитивном, конструктивном ключе. Такой подход замечен как на конференциях и выставках, так и на форумах в сети интернет.

Для представления информации в вычислительных машинах используется двоичное представление, так как его проще реализовать с технической точки зрения. На сегодняшний день на логическом уровне принято

считать, что минимальным участком памяти, к которому возможен доступ, имеет длину 8 бит. Такой участок памяти обычно называется **ячейкой памяти**. На аппаратном уровне размер минимального участка памяти, к которому можно обратится, обычно равен разрядности платформы. Сейчас популярные платформы имеют разрядность 8, 16, 32, 64, 128 и 256 бит. Реальную физическую разрядность называют **машинным словом**, именно столько информации передается в вычислительном устройстве за один такт. Для персональных компьютеров основная разрядность 64 бита. Каждый байт памяти имеет свой номер, который называется **адресом**.

Для разных операций числа записываются в разных видах. Например, существует два вида записи чисел в зависимости от направления расположения ячеек с разрядами, несущими большее значение: **Big ending** и **Little ending**. В первом случае сначала производится чтение байт с меньшими разрядами, во втором сначала читаются большие разряды. В платформах x86, x86-64 корпорации Intel(TM) используется Little ending архитектура. Для записи отрицательных значений используется крайний левый бит, являющийся флагом отрицательного числа.

Для выполнения сложения удобно использовать **прямой код**, т.е. обычную запись числа. Вычитание можно также выполнить как сложение, но для этого необходимо инвертировать каждый бит одного из чисел, что будет равносильно умножению числа на -1, выполнить сдвиг на недостающее количество бит и выставить флаг отрицательного числа. Такое инвертированное число называют **дополнительным кодом**.

Некоторые операции удобнее выполнять в

упакованном десятичном формате. В этом случае необходимо каждое десятичное число представить с помощью 4 бит. Таким образом, в один байт помещается два десятичных числа вместо 256 значений, что увеличивает объем памяти требуемой для хранения значения, но увеличивает скорость выполнения вычислений и позволяет обойти ограничения на максимальное целое значение, ограниченное разрядностью архитектуры.

Действительные числа представляются как **значения с плавающей точкой**. Для этого используется стандартный вид $Mx10^P$, где М — это **мантисса** числа, а Р – его **степень (порядок)**. В вычислительной машине мантисса хранится в нормализованном виде в диапазоне значений 0.0<мантисса<2.0.

В памяти вычислительной машины значение с плавающей запятой хранится в виде двух значений, а значит вычисления с такими данными происходят медленнее. В одном значении хранится знак и значение мантиссы, в другом знак и значение степени. Причем значение степени хранится как **смещенное число**, ставящее в соответствие минимальному значению 0, что позволяет упрощать операции сравнения двух чисел и вычисления с ними.

1.3. Процессы

Компьютеры реализуют две основные сущности: данные и их преобразования, выполняемые процессами. Такой подход стал возможен после того, как в середине XX-го века было предложено хранить данные и программу в одной быстродействующей памяти. В результате программы получили возможность самостоятельно изменять себя в процессе обработки данных.

Базовые операции преобразования данных — это **логические** и **математические** преобразования. Логические операции являются частью теории математической логики, основоположником которой стал немецкий математик Готфрид Вильгельм Лейбниц. Он пытался построить универсальный язык, с помощью которого можно решать споры между людьми путем вычислений. На фундаменте, заложенным Лейбницем, ирландский математик Джордж Буль построил систему вычислений, оперирующих не числами, а высказываниями. Под высказыванием понимается любое утверждение, которое может быть или истинно (соответствует действительности) и равное 1, или ложно (не соответствует действительности) и равное 0. Истина часто выбирается равное 1, т. к. это соответствует «возмущению» системы, повышенному напряжению в канале и т. д., что удобнее при технической реализации. Например, если система работает, более естественно будет восприниматься повышение напряжения в канале связи, а когда система вышла из строя, напряжение скорее всего будет отсутствовать.

Существует всего три основных логических операции, с помощью которых можно реализовать любое логическое преобразование: **НЕ, И, ИЛИ**. Операция НЕ

(отрицание, дополнение, инверсия) называется унарной, т.к. принимает всего один аргумент и инвертирует значение, т.е. **да** изменяется на **нет**. Операция И (конъюнкция, логическое умножение) называется бинарной, т.к. принимает два аргумента и возвращает 1 только в том случае, если оба входных значений равны 1. Операция ИЛИ (дизъюнкция, логическое сложение) также называется бинарной, т.к. принимает два аргумента и возвращает 1 в случае, если хотя бы один аргумент равен 1. Для логических функций независимо от их сложности возможно построение таблиц истинности.

В аппаратном обеспечении по технологическим причинам часто используются дополнительные функции: **исключающее ИЛИ** (штрих Шеффера) и элемент **И-НЕ** (стрелка Пирса). Каждый из этих двух элементов позволяет реализовать все базовые логические функции, а значит для реализации вычислений фактически достаточно использовать один из этих элементов, что и приводит к экономии.

Вторая важная составляющая вычислительных устройств – возможность выполнять арифметические вычисления и прочие математические преобразования. Первыми математическими преобразованиями стали арифметические операции сложения, вычитания, умножения и деления, выполняемые с арабскими цифрами в позиционной системе счисления. Они были разработаны арабским ученым Мухаммедом аль-Хорезми, имя которого в последствии стало нарицательным и дало имя понятию алгоритма. Преимуществом этих операций была возможность производить операции над многозначными числами «в уме», что быстрее, чем при использовании популярных тогда табличек для вычислений, применяемых египтянами. Это стало

конкурентным преимуществом в торговых делах, что можно наблюдать и сейчас на любом восточном базаре, когда продавец пытается непрерывно делать все более «выгодные» предложения, не давая потенциальному покупателю просчитать свою реальную выгоду.

Важность математической составляющей программного обеспечения подтверждается тем, что среди специалистов наивысшего уровня принято называть программы математическим обеспечением. У всех разделов математики есть один общий секрет, они сложны для понимания, но если человек в них разобрался один раз, он сможет его понимать и использовать всю свою жизнь. В связи с этим нельзя игнорировать математическую составляющую программирования, при слабых знаниях базовых разделов математики нужно набраться терпения и разобраться с плохо усвоенными темами раз и навсегда.

Кроме известных всем со школьной скамьи алгоритмов выполнения арифметических операций, при разработке программ приходится на постоянной основе разрабатывать собственные алгоритмы. Алгоритмы изучаются «Теорией алгоритмов» и являются одним из фундаментальных понятий науки «Информатики».

Главным элементом понятия алгоритм является понятие «**исполнитель алгоритма**». Это абстрактное понятие, которое может быть физическим объектом, процессом или командой языка программирования. Главным свойством для него является наличие «**системы команд исполнителя**», т.е. полного набора действий, которые исполнитель алгоритма умеет выполнять. Исполнитель алгоритма не должен задумываться при исполнении алгоритма, должен строго исполнять его инструкции и получить результат, предусмотренный алгоритмом. Требование к

«**формальному**» выполнению алгоритма переносит процесс размышления о возможной последовательности действий с исполнителя на разработчика алгоритма. Таким образом, алгоритм является **формализованным** решением задачи.

Рассмотрим **свойства алгоритмов.**

1. Алгоритм должен разбивать процесс выполнения некоторого действия на последовательность отдельных шагов. Это свойство алгоритма называется **дискретностью**.

2. Алгоритм должен быть ориентирован на конкретного исполнителя алгоритма, которому соответствует определенная система команд исполнителя, поэтому при записи алгоритма допустимо использовать только команды системы команд исполнителя. Это свойство называется **понятностью**.

3. Алгоритм не должен содержать команд, воспринимаемых неоднозначно. Каждая команда при исполнении в разное время разными исполнителями должна давать одинаковый результат. Это свойство алгоритма называется **определенностью**.

4. При исполнении алгоритм должен прекратиться за конечное число шагов и должно быть получено конкретное решение. Вывод о том, что решения не существует, тоже результат. Это свойство называется **результативностью**.

5. Алгоритмы должны обеспечивать решение не одной задачи, а целого класса задач. Это свойство называется **массовостью**.

6. Алгоритмы обладают **эффективностью**. Это свойство описано в пункте 7.3 данной книги.

При разработке алгоритма необходимо прилагать усилия к выполнению свойств 1-4. Если класс задач определить как различные наборы входных значений, то

свойству массовости все алгоритмы будут удовлетворять автоматически.

Алгоритмы могут быть представлены в текстовом, формульном, табличном, графическом виде или на языке программирования. Интересной особенностью является факт отказа признания алгоритмов в текстовом виде в процессе обучения. Это делается для повышения мотивации учащихся в изучении графического представления алгоритмов. В практической деятельности текстовое описание алгоритмов, один из способов написания исходных текстов программ и является одним из двух самых часто используемых наравне с графическим представлением.

Наиболее популярный способ графического представления алгоритмов называется блок-схемой и описан в ГОСТ 19.701-90. **Блок-схема** – это ориентированный граф, в котором используются специализированные вершины графа для обозначения действий алгоритма и определяется порядок их исполнения. В соответствии с теорией графов все вершины обозначаются точкой, в блок-схемах тип наиболее часто используемых действий обозначается различными геометрическими фигурами. Другим отличием блок-схем от графов является ограниченное количество способов отображения ребер графа, которые в блок-схеме называются управляющими линиями. Так достигаются такие преимущества как наглядность и явное отображение управления. Непосредственно способы отображения блок-схем лучше всего изучать по соответствующему регламентирующему документу ГОСТ 19.701-90. Часто, при подготовке документов для внутреннего использования среди ограниченного круга лиц, используются упрощения при рисовании блок-схем. Могут не соблюдаться размеры или пропорции блоков,

вводятся собственные блоки, некорректно отрисовываются управляющие линии. В таком случае указывается, что блок-схема выполнена в общем виде.

Подведем итог, перечислив все возможные преобразования, выполняемые процессами:

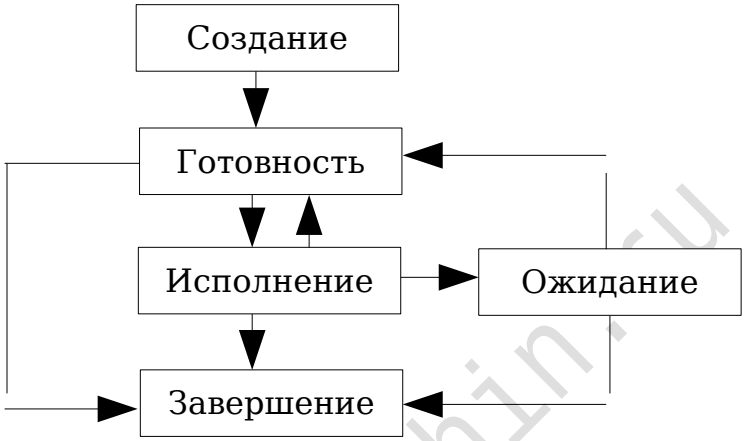
- логические преобразования;
- математические преобразования.

На их основе производится построение алгоритмов, выполняющих все преобразования данных в вычислительной технике.

Современные микропроцессоры, т.е. микросхемы, производящие вычисления в вычислительной технике, ассоциируют понятие процесса с каждой пользовательской или системной программой. Программа является процессом. Понятия «программа» и «процесс» могут использоваться как синонимы, однако под **процессом** чаще понимают набор данных и инструкций для микропроцессора в период исполнения на микропроцессоре. Под **программой** понимается набор данных и инструкций для создания и исполнения в процессе. При этом процесс может находится в следующих состояниях:

- **Создание**, подготовка необходимых устройств и ресурсов для управления процессом и его запуска в микросхеме.
- **Исполнение**, выполнение команд в микросхеме.
- **Ожидание**, отсутствие одного из ресурсов, кроме микропроцессорного времени. Например, данных из файла.
- **Готовность**, все ресурсы подготовлены, но не выделено микропроцессорное время.
- **Завершение**, освобождение всех ресурсов вычислительной машины и, соответственно, прекращение существования процесса.

Взаимодействие этих состояний процесса приведено в [6].



Реализация различных состояний процесса возможна только в рамках концепции **прерываний**. Она подразумевает возможность приостановки исполнения процесса для переключения на более приоритетный и описывает процедуру такого переключения. **Причинами прерываний** могут быть:

- одно из устройств выполнило действие, требующее обработки на микропроцессоре;

- активному процессу требуется некоторый ресурс или действие над ним, т.к. любой ресурс работает медленнее микропроцессора, можно заменить активный процесс на следующий, по очереди требующий исполнения на микропроцессоре;

- активным процессом выполнена операция, не имеющая смысла, например, деление на ноль; операция перевела микропроцессор в состояние, не гарантирующее правильность выполняемых вычислений, например, арифметическое переполнение;

- возникла необходимость синхронизации между процессами, использующими одни данные. Обычно такие

процессы называют параллельными.

Опишем **действия**, выполняемые **при прерывании**:

- получается запрос на прерывание;
- запоминается состояние прерванного процесса;
- передается управление прерывающей программе;
- обрабатывается прерывание;
- восстанавливается состояние прерванного процесса.

Псевдопараллельное исполнение процессов реализуется приемом, называемым **режимом разделения времени**. Его суть заключается в параллельном исполнении нескольких процессов на одном аппаратном микропроцессоре. При запуске каждого процесса определяется степень его важности и, в соответствии с этим, определяется его приоритет. При исполнении каждому процессу выделяется некоторое время для его выполнения в соответствии с приоритетом. В связи с тем, что интервалы времени, на которые выделяется микропроцессор, очень малы, намного меньше времени, требуемого для вывода звука, изображения или воспроизведения управляющего воздействия на исполнительный механизм технической системы, например, электродвигатель, создается впечатление одновременного исполнения нескольких процессов.

При параллельном выполнении процессы изолированы друг от друга, однако они могут осуществлять взаимодействия между собой. Набор межпроцессорных взаимодействий, доступных процессу, определяется операционной системой. Операционные системы будут рассмотрены в следующих разделах. Всего существует три основных **задачи взаимодействия между процессами:**

- **передача данных**;

- **совместное использование** одной копии данных;
- **извещение** другого процесса или нескольких процессов о некотором событии.

В соответствии со стандартом на операционных систем POSIX, задачи взаимодействия между процессами реализуются следующими средствами:

- сигналы;
- каналы;
- именованные каналы;
- сообщения очереди сообщений;
- семафоры;
- сокеты;
- разделяемая память.

Внутри процессов существуют более мелкие единицы выполнения – **потоки**. Они обмениваются информацией внутри процесса без ограничений, но для взаимодействия с другими процессами используют обычные механизмы взаимодействия между процессами. Одно из главных преимуществ потоков — это возможность параллельного исполнения программы на вычислительных машинах с несколькими микропроцессами или на многоядерных микропроцессорах.

1.4. Базовые алгоритмы для программиста на языке Си и способы их применения

В каждой области применения языков программирования можно выделить ряд типовых задач, с которыми приходится сталкиваться наиболее часто. Для языка Си, который часто используется для разработки аппаратнозависимых или наиболее вычислительноэффективных проектов, к таким можно отнести задачи кодирования и шифрования. Если задачи кодирования естественно связаны с разработкой драйверов, преобразованием данных между различными аппаратными протоколами, то шифрование изначально считалось специфическим направлением разработки.

Как указано выше, под кодированием понимается преобразование данных, основанное на правиле, коде. Примером может служить преобразование текста из кодировки UTF-8 в любую другую кодировку. Такое преобразование не должно менять объем передаваемой информации, изменению подвергается только способ её представления. Под **шифрованием** понимается именно сокрытие информации в сообщении, канале связи, либо другим образом. Более подробно шифрование будет рассмотрено ниже.

Наиболее простым видом кодирования является **контрольная цифра**. Правило его применения заключается в дописывании в конце числа цифры 1, если сумма единиц в двоичном представлении числа нечетно, или 0, если сумма единиц четная. Например, в конце числа, представленного в двоичном виде 1000101, необходимо дописать **1**, чтобы количество единиц стало 4, т.е. четным. Получается 1000101**1**. В числе 1001011 необходимо дописать **0**, т.к. число единиц уже четно. Получается 1001011**0**. Алгоритм применения

контрольной цифры — наиболее быстрый способ проверки ошибок в передаваемых данных. Контрольная цифра обычно применяется при разработке аппаратных протоколов обмена данными и служит для нахождения нечетного количества сбойных бит. Она позволяет определить наличие сбоя в одном, трех, пяти и так далее бит.

Применение обобщения алгоритма контрольной цифры к десятичным и другим числам называется **контрольной суммой**.

Контрольная сумма — это значение, рассчитанное по определенному правилу и используемое для проверки правильности переданного числа. Правило применения контрольных сумм основывается на сравнении числа, полученного путем вычислений, проводимых с переданными значениями, включая число контрольной суммы и некоторой константы. В этом случае контрольная сумма дополняет результат вычислений до константного значения.

Приведем простейший пример контрольной суммы. Пусть сумма цифр числа должна всегда быть равна 100. Тогда контрольной суммой для числа 263567, будет равно 100-(2+6+3+5+6+7)=71. Способ передачи контрольной суммы обычно сложнее передачи контрольного числа, а значит его применение возможно не во всех аппаратных реализациях.

Контрольные суммы применяются в сетевых протоколах, файловых системах, архиваторах и т.д. Наиболее популярными являются **CRC8**, **CRC16** (CRC – циклический избыточный код) и **CRC32**. Хотя CRC чаще относят к алгоритмам хеширования. Контрольные суммы могут применяться независимо от самих данных. Основным требованием для контрольных сумм является отличие между наиболее близкими значениями, т.е. чем

меньше разница двух чисел, тем больше должны отличаться их контрольные суммы. Таким образом, контрольные суммы позволяют находить ошибки, состоящие из подряд идущих неверных значений.

Хеширование, хеш-функция, хеш-код, функция свертки – это наиболее общее название преобразований входного массива данных произвольной длины в выходную битовую строку фиксированной длины. Выходная строка обычно называется **хешем**. Хеширование применяется для сравнения массивов данных. Например, выполняется получение хеша файла. Через некоторое время хеширование повторяется и полученный хеш сравнивается с более ранним. Таким образом можно узнать, изменилось ли содержимое файла за прошедшее время. Примеры алгоритмов хеширования: **MD5**, **MD6**. В РФ принят алгоритм хеширования «Стрибог», закрепленный в **ГОСТ Р 34.11-2012** (**RFC 6986**).

Необходимо понимать очевидную вещь: хеш обычно содержит намного меньше цифр, чем исходный массив, поэтому одному хешу соответствует множество массивов такой же длины. Такая ситуация называется **коллизией**. Причем чем длиннее массив, тем больше массивов такой же длины имеют такой же хеш. Однако не следует забывать, что чем длиннее массив, тем больше общее количество таких массивов. В результате, при увеличении длины массива вероятность совпадения хешей все равно остается примерно одинаковой. Правило, применимое для контрольных сумм, которое говорит о максимальном различии хешей ближайших массивов данных, распространяется на все хеш-функции.

Хеширование применяется для проверки соответствия некоторого сообщения другому сообщению. Одно из применений хеш-функций — поиск ошибок в

передаваемых данных. Другим способом применения является ускорение поиска данных. Хеш содержит фиксированное количество символов, например, в MD5 это 32 символа. Посчитав хеш от 100 страниц текста, можно сравнить 100 страниц с другим текстом, также посчитав хеш от второго текста. Для двух текстов преимущество в производительности будет неочевидно. Однако, если текстов десятки, достаточно будет сравнивать 32 символьные числа, что на порядки ускорит время поиска.

Наиболее сложные алгоритмы преобразования данных изучаются наукой – **криптография**. Эта наука ведет свою историю несколько тысяч лет и изучает математические методы обеспечения конфиденциальности и аутентичности информации. Под конфиденциальностью понимается невозможность прочтения информации посторонним, а под аутентичностью — целостность и подлинность авторства, невозможности отказа от авторства. Криптография не занимается изучением защиты от обрыва связи, влияния социальных угроз информации.

Базовыми понятиями в этой области являются:

- открытый (исходный) текст;
- шифрованный (закрытый) текст;
- криптосистема (шифр);
- ключ;
- принцип Керкгоффса;
- расшифрование;
- дешифрование (“взлом”);
- криптоанализ;
- криптоаналитик;
- криптографическая атака;
- абонент криптографической системы;
- взлом;

- имитозащита;
- имитовставка;
- цифровая электронная подпись.

С целью выработки навыка самостоятельного поиска информации их объяснение приведено не будет. Значение этих терминов легко найти в сети Интернет.

За время существования криптографии предпринимались попытки создания принципиально не вскрываемых шифров, например, **шифр Вернама**, который заключается в двоичном сложении исходного текста с однократно используемой случайной последовательностью. Однако построение универсальной системы, решающей все проблемы, связанные со скрытой передачей информации, еще не увенчались успехом.

В зависимости от способов использования ключей, все методы делятся на три вида.

Тайнопись. Посторонним лицам неизвестен алгоритм шифрования и доступен только отправителю и получателю. Данный способ противоречит принципу Керкгоффса и на сегодняшний день используется редко.

Симметрическое шифрование / шифрование с закрытым ключом. Используется общеизвестный алгоритм шифрования. Ключ, одинаковый для шифрования и расшифрования, известен только отправителю и получателю. Ключ выбирается до начала обмена шифрованной информацией и сохраняется до завершения обмена.

Ассиметрическое шифрование / шифрование с открытым ключом. Используется общеизвестный алгоритм шифрования, ключ для шифрования (**открытый ключ**) предоставляется в открытом доступе. Ключ для расшифрования (**закрытый ключ**) известен только получателю. Сеанс обмена шифрованной информацией инициируется получателем путем

генерации ключа шифрования, разбиения его на открытую и закрытую часть и опубликования своего открытого ключа. Зашифрованная с помощью открытого ключа информация может быть прочитана только получателем, т.к. только он обладает закрытым ключом.

Система шифрования может реализовываться только программными средствами или с помощью программно-аппаратного комплекса.

Комплекс средств вокруг способа симметричного шифрования может включать вспомогательные функции для управления ключами, реализацию самого алгоритма шифрования, другие сервисные функции. К достоинствам симметричного шифрования относится скорость шифрования, простота реализации, меньшая длина ключей и изученность. Недостатки – сложность обмена ключами между абонентами и сложность управления ключами при большом количестве абонентов. Данный способ шифрования невозможно использовать для подтверждения авторства, т.к. ключ известен всем абонентам.

Преимущество ассиметрического шифрования состоит в отсутствии необходимости передачи секретного ключа. Недостатками является потребность в значительно больших вычислительных ресурсах, чем при симметрическом шифровании, а также необходимости обеспечения отличия между источниками шифрованной информации, для чего часто используются сертификаты. Не менее серьезным недостатком является невозможность скрыть источник и получателя сообщения.

С одной стороны шифрование — это в первую очередь сложная математика, с другой — технология, которой мы пользуемся каждый день в сети интернет. Приводить примеры математических преобразований для

шифрования/расшифрования не имеет смысла без глубокой математической подготовки, поэтому сосредоточимся на технологиях, которые их используют.

Любая технология шифрования может стать уязвимой при её неправильном использовании. Как сказано выше, криптография не изучает проблемы психологических особенностей пользователей, применяющих криптографические технологии и межчеловеческие деструктивные отношения, такие как обман, предательство, шантаж. Чаще всего именно эти факторы приводят к снижению эффективности применения шифрования. Также существуют технологические факторы, понижающие эффективность шифрования. Например, виртуальная файловая система шифрования информации **encfs** надежно шифрует данные, однако подвержена атаки наблюдения за файловой системой. Другими словами, если злоумышленник имеет доступ к зашифрованным данным, он, наблюдая за изменениями в файлах, может значительно упростить взлом зашифрованной информации. Человеческий фактор чаще всего приводит к следующим факторам ухудшения надежности шифрования:

- выбор пароля, не обеспечивающего надежное шифрование. Слишком короткий, использует малый набор символов, использование в пароле слов, памятных дат и т.д.

- неверная настройка системы шифрования. Выбор алгоритмов шифрования, не соответствующих степени важности информации. Неумение правильно настроить систему шифрования. Недостаточная квалификация сотрудников.

- Недостаточная благонадежность сотрудников. Неумение хранить секреты, намеренное вредительство,

предательство.

Приведеное выше описание призвано подчеркнуть важность и сложность поднятой проблемы, однако её изучение требует высокой квалификации.

Администраторы компьютерных сетей и разработчики чаще всего используют технологию, содержащую шифрование под названием **SSH** (защищенная оболочка). Технология основана на одноименном сетевом протоколе SSH-2 (RFC 4251), позволяющем передавать файлы и управлять операционной системой вычислительной машины. Протокол шифрует весь передаваемый трафик, создавая безопасный канал почти для любого другого сетевого протокола. Позволяет передавать звуковые и видеоданные в масштабе реального времени, выполнять сжатие передаваемых данных для запуска «удаленных рабочих столов».

В некоторых странах (Франция, Россия, Ирак и Пакистан) до сих пор требуется специальное разрешение в соответствующих структурах для использования определенных методов шифрования, включая SSH. Однако, по-факту, в РФ это требование не выполняется.

Технология SSH построена по архитектуре клиент-сервер. Для подключения к другой вычислительной машине необходимо установить программу-клиент SSH, соответственно, на вычислительной машине, к которой производится подключение, должна быть установлена программа-сервер SSH. На одной и той же вычислительной машине могут быть одновременно установлены и программа-клиент и программа-сервер.

Непосредственно после установки SSH-сервера необходимо выполнить ряд действий для повышения надежности зашифрованного канала, а если SSH-сервер используется в общедоступной сети, включая интернет,

необходимо выполнить скрытие наличия SSH-сервера у этого компьютера.

Основные алгоритмы шифрования, о которых необходимо знать, являются **DES**, **Triple DES**, **AES**, **ГОСТ 28147-89**. В этом списке Triple DES и ГОСТ 28147-89 содержат в себе по несколько алгоритмов шифрования. DES считается устаревшим, однако на его примере проще понять механизм шифрования.

Последним классом алгоритмов, о которых будет упомянуто в этом разделе — **сжатие данных**. В разное время популярность имели разные алгоритмы сжатия. Они обычно сложны для понимания и их разработкой занимаются математики. Каждый из алгоритмов дает максимальное значение коэффициента сжатия данных определенных форматов, которые были популярны в период создания алгоритма. Во времена наибольшей распространенности файлов с текстом, чаще использовался алгоритм сжатия **RAR** (Архив Рошаля). Разработанный Филом Кацем алгоритм **ZIP** (RFC 1951) лучше сжимает данные графических файлов и наборы файлов с разнотипным содержимым. Алгоритм базируется на алгоритме Хаффмана и LZ77 и позволяет его свободно использовать, что стало причиной его популярности в сети интернет. Следующим алгоритмом, получившим широкое распространение в компьютерных сетях, стал **7Z** (архиватор 7-zip), основанный на **LZMA**. Его активно применяют как в сетевых устройствах, так и для сжатия мультимедийной информации.

Основной принцип сжатия называется **механизм кодирования совпадений**, заключается в поиске повторений и обозначение их более короткой последовательностью, которая не встречается в файле. В результате экономия равна разнице между размером повторяющейся последовательности в файле и

последовательности, используемой для её замены, умноженной на количество повторений. Таким образом лучше всего сжимаются файлы, состоящие из многократно повторяющихся длинных последовательностей. Другое следствие из этого правила заключается в том, что зашифрованные файлы не сжимаются, т. к. практически не содержат повторяющихся последовательностей.

Другой базовый принцип для алгоритмов сжатия — **принцип скользящего окна**. Смысл этого принципа заключается в наличии некоторого окна (буфера), в котором содержится информация из участков потока данных, встреченных ранее, и благодаря этому возможно повышать степень сжатия.

Следует учитывать, что современные графические и мультимедийные форматы уже содержат в себе алгоритмы сжатия данных, поэтому на текущий момент пользователи персональных вычислительных устройств стали намного реже использовать программы сжатия данных с помощью архиваторов. За них это делают большинство популярных программ.

1.5. Базовое программное обеспечение, операционные системы

Программное обеспечение является частью почти любого аппаратного устройства внутри вычислительного устройства. Программное обеспечение этого класса имеет сленговое название — **прошивка** устройства. Они используются в накопителях информации, сетевых картах, устройствах позиционирования (мышь, контроллер сенсорного экрана). Центральный микропроцессор содержит в себе настолько сложное программное обеспечение, что его уже относят к классу операционных систем.

Управляет работой всех устройств программно-аппаратное обеспечение UEFI — интерфейс расширяемой прошивки. Это класс программ, часть из которых содержат в себе специальные программы управления устройствами – драйвера. Программное обеспечение этого класса не поддается управлению и обслуживанию в домашних условиях, поэтому ошибка в нем делает устройство неработоспособным.

Программы (прошивки), встроенные в аппаратное обеспечение, загружаются первыми. Не следует их путать с драйверами для операционных систем. Следующим этапом выполняется загрузка UEFI, которая проводит диагностику компонентов вычислительной машины.

Говоря о программном обеспечении этого уровня, необходимо вспомнить технологию **Plug and Play**. До её появления устройства внутри вычислительной машины могли сломаться из-за их неправильного подключения. Технология Plug and Play содержит три основных компонента: операционную систему, поддерживающую Plug and Play, Plug and Play BIOS/UEFI и Plug and Play -

устройства с соответствующими драйверами. Поэтому полное решение проблемы Plug and Play требует поддержки как на программном, так и на аппаратном уровнях. Сегодня название этой технологии стало именем нарицательным и используется, когда говорят о наличие в устройстве элементов самодиагностики, которые в совокупности с технологией подключения способны предотвратить поломку устройства при неправильном подключении.

Далее UEFI запускает программу начальной загрузки операционной системы (**загрузчик операционной системы**), который способен предоставлять выбор загружаемой ОС. Наиболее популярным является загрузчик ОС WindowsTM, который пригоден только для загрузки операционной системы семейства WindowsTM. Мультизагрузчик **grub** позволяет загружать большое количество типов и версий операционной системы.

После выбора операционной системы происходит загрузка основной программы вычислительной машины для пользователя - **операционной системы (ОС)**. Существует множество устройств, которые разработаны без использования ОС. Обычно это простые автоматизированные системы, такие как модели на радиоуправлении, дверные электро-механические замки. Для более сложных устройств, включая встраиваемые, необходимо применение ОС. ОС применяют в случаях, если необходимо распределять полномочия между программами, предоставлять ресурсы единого формата разным программам, например в виде библиотек функций.

Распределение полномочий между программами и пользователями позволяет решать следующие задачи:

- решать разные задачи на одной вычислительной

машине;

- позволять задачам сохранять результат или обмениваться результатами своей работы с другими задачами;
- разделять полномочия между программами, ресурсами и пользователями;
- имитировать режим разделения времени, реализовывать виртуализацию или другую технологию;
- предоставлять пользователю возможность управлять процессами и программами.

Под упомянутым ранее термином **ресурс** подразумевается обобщение, которое может быть и аппаратным устройством и функциональным блоком. Кроме того часто используется понятие **виртуальный ресурс**, являющийся моделью физического ресурса, реализуемой за счет другого физического ресурса. Например, образ оптического диска (CD, DVD, BlueRay) может быть файлом на носителе информации, но ОС будет его отображать как физическое устройство. Более популярным примером является виртуальная оперативная память, которая реализуется файлом на носителе информации, и позволяет частично записывать информацию оперативной памяти в файл. Таким образом объем «дорогого ресурса» увеличивается за счет более дешевого носителя информации.

Функции ОС описаны в стандарте POSIX ISO/IEC/IEEE 9945:2009 (рекомендуется читать как «позикс»). В нем содержится определение около 1000 системных вызовов и несколько сотен команд оболочки и утилит ОС, но отсутствуют способы их реализации. Кроме графических интерфейсов, ОС может включать набор средств разработки. Стандартизация ОС упрощает взаимозаменяемость разных ОС, упрощает перенос прикладного программного обеспечения между

различными ОС, включая полную совместимость на уровне исходных кодов. Самым заметным результатом существования стандарта ОС стало эффективное разворачивание Интернета в 90-х годах.

Современные операционные системы на базе ядра Линукс настолько компактны, что используются в сетевых маршрутизаторах, телевизорах, мобильных устройствах, дверях, пылесосах, рюкзаках, куртках.

Считается, что создание языка Си и системы UNIX — две самые важные вехи в развитии компьютерной индустрии. Си — первый мультиплатформенный язык, а UNIX, так как написана на нем, — первая мультиплатформенная ОС [7].

Язык Си развивался параллельно с операционной системой Unix, поэтому язык Си исключительно ценен с практической точки зрения. С одной стороны он содержит в себе инструменты для работы со всеми элементами ОС, с другой не привязан ни к одной ОС. Все взаимодействия с ОС вынесены в подключаемые модули. В связи с этим для переноса исходного текста программы в другую ОС требуется только пересборка подключаемых модулей.

Важность ОС Unix в том, что в процессе её развития были опробованы многие технологии жизненного цикла программного обеспечения, был заложен фундамент для первого стандарта для ОС POSIX, который до сих пор является основным стандартом для всех разработчиков ОС. Впоследствии Unix дала жизнь проекту, из которого появилось ядро ОС Линукс. С помощью краткого исторического обзора предпримем попытку показать взаимосвязь развития операционных систем, языка Си и информационных систем как единого процесса.

Первая версия ОС была задумана и реализована Кеном Томпсоном при участии Денниса Ричи и Брайана

Кернигана в 1969 году. Отметим, что Деннис Ричи и Брайан Керниган — авторы языка Си.

Уже тогда ОС имела следующие черты:

- простая метафорика, состоящая всего из двух базовых понятий: вычислительный процесс и файл;
- имела компонентную архитектуру, т. е. выполнялся принцип «одна программа — одна функция», а также средство связывания разных программ, названное оболочкой;
- использовался принцип минимизация ядра ОС и количества системных вызовов;
- была независимой от аппаратной архитектуры;
- имела файловую систему и унификацию форматов записи и чтения файлов;
- обеспечивалась возможность использования системы двумя пользователями;
- имела командный интерпретатор.

В 1971 году был выпущен первый комплект документации, после которого родилась традиция объявлять о выходе новой версии после выхода новой версии документации. Таким образом был нарушен один из основных принципов инженерной работы, требующей сначала разработки документации, а уже затем создание самого продукта.

В 1972 году в ОС появились программные каналы (pipes).

В 1973 году ОС была переписана на языке Си и уже имела 25 установок.

В 1974 году была опубликована историческая статья "UNIX Timesharing Operating System" в журнале Communications of the ACM. Эта статья и сейчас свободно доступна в сети интернет. Значительная её часть до сегодняшнего дня актуальна как концептуально, так и на уровне реализации. Компания Bell Labs объявила о

возможности бесплатного получения исходных текстов Unix для образовательных целей. Такой подход стал прообразом для движения **open source,** основанного на лицензии **GNU**.

В 1975 году Си уже имел легкопереносимый компилятор, таким образом впервые была решена проблема переносимости программ на уровне исходных кодов.

В 1976 году была создана библиотека стандартного ввода/вывода (stdio), используемая до сих пор. В этом же году появился «дистрибутив» UNIX под названием Berkeley Software Distribution (**BSD**), в котором впервые был реализован стек транспортных протоколов TCP/IP (Transport Control Protocol/Internet Protocol). Эта работа финансировалась министерством безопасности США.

В 1977 году происходила смена 16-ти разрядной архитектуры на 32-х разрядную. В связи с этим в язык Си были введены новые типы данных: union, short integer, long integer и unsigned integer.

В 1978 году была организована группа поддержки (UNIX Support Group — **USG**), которая по-сути является прообразом всех форм поддержки пользователей всех программ и групп разработки стандартов в области информационных технологий. В этом году Microsoft Corporation совместно с Santa Cruz Operation (SCO) произвели вариант UNIX под названием XENIX.

В 1983 году была выпущена версия UNIX, к которой производитель обещал поддержку **для всех последующих версий**. Данный подход показал свою **нежизнеспособность**. Все новшества в этом году были связаны с активным внедрением хеш-таблиц, т.е. таблиц для быстрого доступа к данным и хешированием данных. Появились семафоры, очереди сообщений и разделяемая память.

В этом же году был выпущен один из первых компьютеров с графическим пользовательским интерфейсом Lisa от компании Apple, который не стал популярным из-за высокой стоимости и плохой надежности аппаратного обеспечения. Это событие стало поводом для применения методов социальной инженерии в конкуренции информационных систем. В среде пользователей DOS TM компании MicrosoftTM распространялись насмешливые прозвища графического интерфейса пока велась работа над ликвидацией технологического отставания.

В 1984 году была выпущена версия названная UNIX System V Release 2. В нем появились новые механизмы работы с памятью, возможность блокировок файлов и записей. На тот момент было выполнено более 100000 установок ОС UNIX.

В 1987 году была выпущена версия UNIX System V Release 3. В ней появились межпроцессные взаимодействия, разделения удаляемых файлов (Remote File Sharing - RFS), разделяемые библиотеки. Было произведено 750000 установок и более 4,5 млн. пользователей.

В эти годы ходили слухи о внедрении в исходные коды ОС DOS и Windows умышленной ошибки, не позволявшей работать электронным таблицам пакетов Lotus продолжительное время без ошибок. На тот момент законодательства регулирующего область разработки программного обеспечения не было, и такие слухи, в совокупности с непредсказуемыми зависаниями ОС при использовании пакетов Lotus, казались вполне правдоподобными.

Начиная с 1990 года популярность продуктов компании WindowsTM отодвинули ОС UNIX и производные от ней ОС на второй план до появления

проекта ОС Линукс в 1994 году. **Линус Тордвальдс** породил массовое движение за открытый исходный код программ. Это движение позволило огромному количеству, участников проекта, точное количество которых не знает никто, создавать подавляющее большинство новшеств именно в этом проекте и только затем они переносятся в другие ОС. После переноса ОС Линукс на мобильные устройства в виде ОС Андроид и iOS, UNIX подобные системы вернули себе лидерство по количеству установок.

Язык Си являлся центром развития информационных технологий на протяжении длительного времени и его возможности развивались, исходя из реальных потребностей разработчиков на основе экспериментов при создании ОС.

ОС Линукс является ядром операционной системы, на основе которого создают наборы программ, именуемые дистрибутивами ОС Линукс. Каждый дистрибутив имеет свою философию развития и направлен на определенную группу пользователей.

В этой книге будет использоваться дистрибутив **АльтЛинукс**. Адреса проекта в сети Интернет http://www.altlinux.ru. Кроме бесплатности, он обладает рядом преимуществ. Не требователен к ресурсам и может эксплуатироваться на моральноустаревших ПЭВМ. Его можно развернуть за 40 минут или установить за час со всеми приложениями. Обновление редко занимает больше 5 минут, при этом обновляются сразу все приложения. После однократной установки у пользователей этой ОС обычно не возникает необходимости её переустанавливать. Команда проекта устойчиво развивается на протяжении многих лет и максимально приближена к российским реалиям. Она выпускает стабильные версии ОС, но есть возможность

получать наиболее свежие обновления, которые могут содержать ошибки. В этом случае приходится в ручном режиме возвращаться к более ранним версиям. Дистрибутив имеет собственный форум, расположенный по адресу http://forum.altlinux.org, на котором можно получить помощь в оперативном режиме.

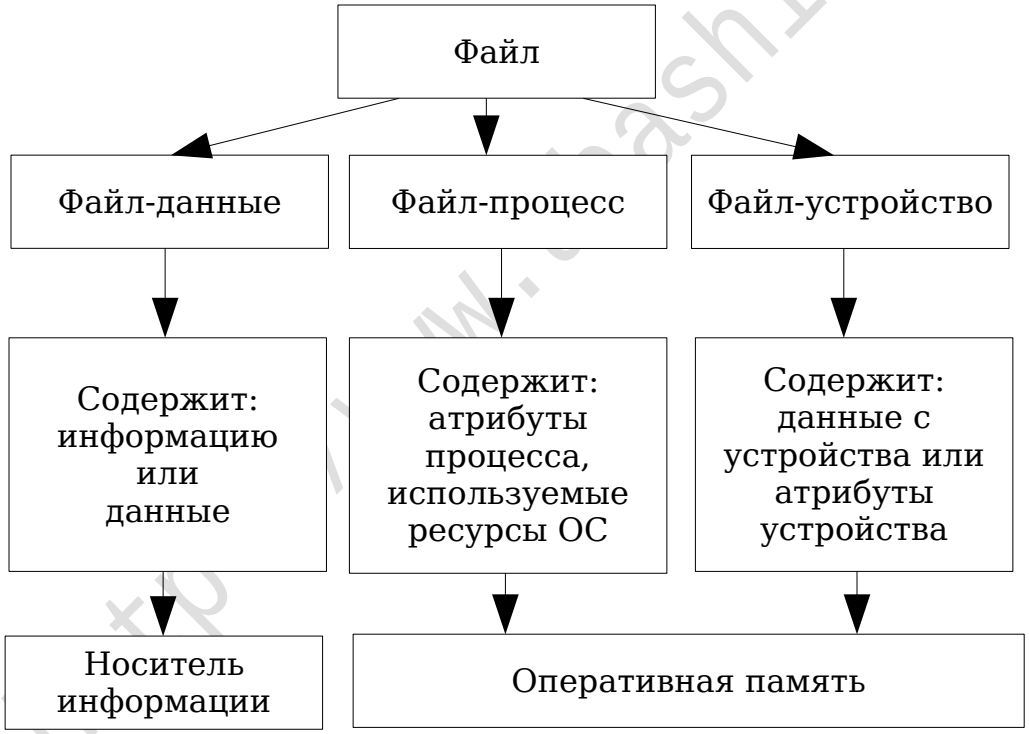
К недостаткам ОС Линукс можно отнести отсутствие драйверов на наиболее дорогие аппаратные комплектующие, выпущенные менее года назад. Однако это может быть и плюсом, не требуется тратить много денег на самое свежее аппаратное обеспечение. Оптимальным является выбор аппаратного обеспечения, выпущенного 2-3 года назад. К этому времени цена на некоторые комплектующие может снизиться до двух и более раз, а их производительность позволит не заниматься обновлением аппаратной составляющей до пяти лет. Решением проблемы совместимости аппаратного и программного обеспечения является поиск информации о поддержки выбираемого аппаратного обеспечения текущей версией ОС до покупки аппаратного обеспечения.

Часто возникает противоречие, связанное со сложностями при использовании новейшего оборудования и дистрибутивов на базе ОС Линукс. Обычно наиболее производительные конфигурации используются для узкопрофильных задач. Такие устройства потребляют большое количество энергии и набирать на них исходные тексты программ экономически не эффективно. Другой отговоркой от покупки 2-3 летнего аппаратного обеспечение является невозможность запуска компьютерных игр со сложными эффектами. Для них уже много лет успешно используются компьютерные приставки, производительность которых не ниже «игровых

конфигураций» ПЭВМ, но стоимость часа игры на приставке, состоящей из стоимости приставки, игры, потребляемой электроэнергии, ниже, чем постоянная эксплуатация «игровой ПЭВМ». Главным преимуществом игровой приставки является отсутствие графических эффектов «подвисания изображения», «несвоевременная прорисовка», «малая дальность прорисовки» и прочие.

1.6. Файловая система

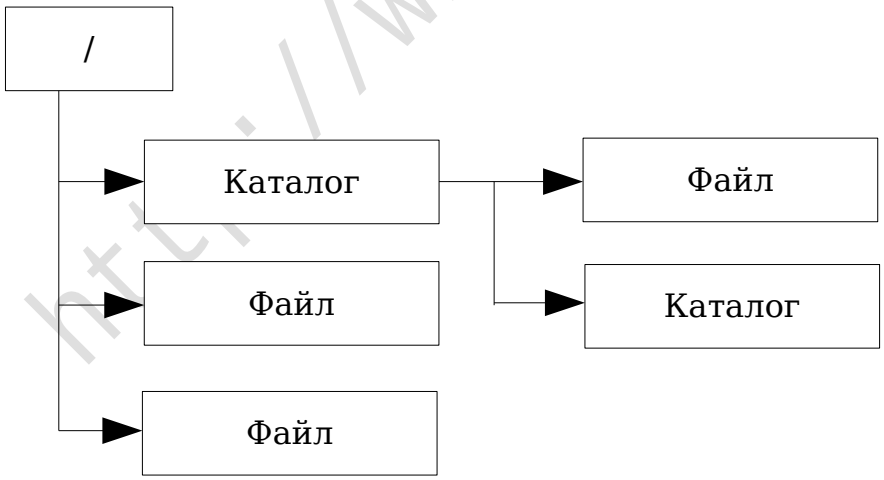
Основное понятие файловой системы – файл. Наиболее общее определение файла следующее: **файл** – именованная последовательность байт. Таким образом, под файлом может пониматься модель любого устройства, например клавиатуры или мыши, байты на носителе информации, сетевое соединение или даже микропроцессор. Файл является упаковкой как для базовой сущности — информация, базовой сущности процесс, а также для устройств.



В случае файла с информацией, информация обычно храниться в устройстве по частым в произвольном порядке, что позволяет экономить физическое пространство накопителя, но приводит к **фрагментации файлов**. В случае процесса, файл процесса является виртуальным и не записывается на носитель

информации. Файл устройство обычно является областью оперативной памяти, через которое производится обмен информацией с устройством. Файлы устройства и файлы процессов существуют только в ОС и содержат конфигурационную информацию, информацию о текущем состоянии процесса/устройства, списки ресурсов, предоставленные ОС данному процессу или устройству.

Для управления файлами используется файловая система. **Файловая система** (ФС) - состоит из драйверов и программ и используется для управления файлами. Благодаря файловой системе работа с файлами производится с помощью унифицированных действий и не требует знания особенностей реализации конкретного устройства. ФС обрабатывает ошибки, возникающие в процессе работы с файлами. Вторым после понятия файл, наиболее важным понятием файловой системы является каталог (папка). **Каталог** — файл специального формата, содержащий записи о других файлах и каталогах и их атрибутах.



Используя каталоги, файловая система реализует иерархическую структуру расположения файлов и каталогов. Именно в каталогах хранятся названия

файлов и вложенных каталогов, общий размер файлов. Каталог, из которого начинается ветвление файловой системы, называется **корневой каталог** и обозначается «/». В операционных системах семейства WindowsTM для просмотра файловой системы используется программа «Проводник», в которой корневой каталог скрыт и вместо него отображается программа «Мой компьютер» или «Компьютер».

Для отображения расположения объекта файловой системы используется понятие путь к файлу или каталогу.

Пути бывают двух типов.

1. **Относительный путь**, содержит перечисление каталогов от текущего каталога до искомого файла, например: groff/man.local .

2. **Полный путь**, содержит перечисление всех каталогов до искомого файла, например: /etc/groff/man.local . Полный путь всегда начинается с символа «/».

Базовый принцип работы с файлами описывается так: **открыл – чтение/запись – закрыл**. Это очень напоминает алгоритм наших действий с кастрюлями на кухне. Мы открываем крышку одной кастрюли (открыл) смотрим в её содержимое или черпаем оттуда суп (читаем) и закрываем крышку (закрыл). Также и с записью, сначала нужно открыть крышку чистой кастрюли, затем налить туда молоко, затем закрыть.

Во время одного сеанса работы с файлом возможно

выполнение чтение и запись в один файл, однако все операции должны выполняться по очереди, а не одновременно. В некоторых случаях допускается одновременное чтение из одного файла, но если выполняется запись, все операции чтения должны быть прекращены и сброшены, т.к. содержимое файла было изменено.

Для работы нескольких пользователей на одной вычислительной машине необходимо иметь возможность разграничивать их права. Исторически сложилась система, в которой все права разделяют на три группы: **владелец** (user); **группа владельца** (group); **остальные** (other). Удобство такой системы заключается в возможности работать с файлами индивидуально, обмениваться информацией внутри группы или предоставить общий доступ к файлу.

Для разграничения доступа различных программ ОС реализуют три вида доступа к программе: **чтение** (read), **запись** (write), **исполнение** (execute). Причем права разграничения доступа процессов можно задавать для каждой группы пользователей. Подчеркнутые буквы английского написания прав используются как общепринятый способ указания прав. Отсутствие права отображается символом «-». Права доступа записываются в следующем порядке: права владельца, права группы владельца, права всех остальных. Приведем примеры:

rwxrwxrwx — владелец имеет доступ на чтение, запись, исполнение этого файла; группа владельца имеет доступ на чтение, запись, исполнение файла; все остальные имеют права на чтение, запись и исполнение файла;

rwxr-x--x — владелец имеет только право на чтение, запись и исполнение файла, группа владельца имеет право на чтение и исполнение файла, остальные имеют

право только на исполнение файла.

Такой способ задания прав удобен, если владелец файла создает текстовый файл, являющийся скриптом. Скрипты имеют возможность исполняться как программы. Группа владельца может просматривать исходный код скрипта и исполнять его, оказывая таким образом помощь владельцу файла. Все остальные могут только исполнять файл, не зная исходный код программы.

Для файла можно задать любую другую комбинацию прав, включая комбинацию, при которой у всех отсутствует любой доступ к файлу. Чтобы понять как исправить такую ситуацию, необходимо вспомнить, что все атрибуты файлов хранятся в каталоге. Удалив все права к файлу, пользователь теряет к нему доступ, но если сохраняются права к каталогу, в которой он находится, значит эти права можно вернуть. После этого можно получить доступ к файлу во вновь заданных режимах.

Реальная система управления файлами немного сложнее. В современные ОС используются так называемые л**ипкие биты**, которые позволяют запускать программы от имени других пользователей, более детальную информацию о них можно узнать с помощью поисковых систем сети интернет. Кроме того, пользователь root (администратор/системный пользователь) имеет доступ на чтение и запись ко всем объектам файловой системы независимо от выставленных прав, однако не имеет прав на исполнение программ, если они не заданы явно. Права к файлу иногда задаются с помощью трех цифр, что менее удобно для начинающих. Способ задания прав доступа не влияет на результат.

Дополним список объектов ФС:

- **жесткая** ссылка. Это запись внутри каталога, которая указывает на другую запись. Она может использоваться только в пределах одного устройства;

- **символьная** или мягкая ссылка. Это файл на диске, указывающий на другой файл или несколько файлов в другом каталоге, которые могут находиться на другом устройстве;

- **канал** (FIFO, PIPE) — файл, расположенный в оперативной памяти, используемый для обмена информацией между процессами;

- **сокет**. Это файл, расположенный в памяти и используемый для обмена информацией между процессами. Обычно сокеты используются для межсетевых взаимодействий по протоколам TCP/IP, однако сокеты являются стандартным способ обмена информацией между любыми процессами.

Файловая система отображается для пользователя как единая иерархическая структура, однако каждая из них умеет работать только с одним типом объектов: файлами с данными; файлами процессами или файлами устройствами. Существуют ФС следующих типов:

- **devfs**. Предоставляет доступ к устройствам компьютера;

- **proc**. Позволяет получить доступ к атрибутам исполняемых в данный момент процессам, а также к параметрам ОС;

- **ext2, ext3, ext4**. Наиболее популярная файловая система для управления файлами с информацией;

- **ReiserFS**. Журнализируемая ФС, ориентированная на хранение множества файлов малого размера, например html-страниц. В ней фрагментация файлов в носителе почти всегда находится на пренебрежимо малом уровне. Может рассматриваться как специализированная ФС для управления файлами с

информацией;

- **VFS**. Виртуальная ФС. Выполняет функцию прослойки между любой ФС, включая FAT32, NTFS, Ext2 и ядром ОС;

- **NFS**. Сетевая ФС. Позволяет реализовывать разделение одной ФС между несколькими компьютерами. В этом случае, каталог ФС может оказаться носителем информации на другой вычислительной машине.

Для добавления одной ФС в другую используются операции **монтирования** и **размонтирования**. Для монтирования ФС достаточно вызвать функцию **mount** и указать её каталог, в которую будет произведено монтирование и путь к монтируемому каталогу или путь к монтируемому устройству. В случае монтирования в каталог уже содержащий объекты, имеющиеся объекты будут скрыты, но не удалены. После размонтирования предыдущее содержание каталога станет вновь доступным. Более подробно вопрос монтирования будет рассмотрен в главе, посвященной администрированию ОС.

Приведем **структуру файловой системы** дистрибутива АльтЛинукс с описанием их содержимого. Список каталогов корневого каталога «/» может меняться от версии к версии, однако имена большинства каталогов будут совпадать.

bin — содержит наиболее важные системные прикладные программы, которые доступны в любых режимах работы ОС.

boot — содержит образы ядер ОС Линукс разных версий и веток, загрузчик, утилиту для тестирования оперативной памяти **memtest**, которая запускается без загрузки ОС, прочую информацию, необходимую для загрузки. Для доступа в этот каталог необходимо обладать правами администратора (root).

dev — предоставляет доступ к устройствам вычислительной машины. Доступ на чтения содержимого этого каталога есть у всех пользователей. Для работы с устройствами требуется обладать правами администратора.

etc — содержит файлы конфигурации ОС, драйверов, служб и прикладных программ. Доступ на чтение предоставляется всем пользователям. Изменять информацию в этом каталоге может только пользователь с правами администратора.

home — содержит каталоги, имена которых соответствуют системным именам пользователей, например для пользователя valerii должен существовать каталог /home/valerii . Доступ к содержимому каталога пользователя имеет сам пользователь и администратор.

lib — содержит библиотеки, необходимые для работы системы, чаще связанные с аппаратным обеспечением.

lib64 — содержит архитектурнозависимые системные библиотеки для архитектуры **x86-64**, т. е. для вычислительных машин, оснащенных микропроцессорами, имеющими 64-х разрядные регистры и команды для логических и арифметических операций.

lost+found — этот каталог используется ФС для исправления ошибок в самой себе. Каталог с таким названием может встречаться в других каталогах.

media — в этот каталог монтируются (присоединяются или добавляются) переносные устройства, такие как оптические дисковые носители, носители информации на основе флеш-памяти и прочее. В случае их успешного монтирования в этом каталоге появляется каталог соответствующий определенному устройству.

mnt — каталог для монтирования дополнительных

накопителей информации, постоянно присутствующих в ОС.

opt — каталог для хранения настроек, которые относятся к конфигурированию системы. В дистрибутивах АльтЛинукс обычно пустой.

proc — каталог, соответствующий ФС, предоставляющей доступ к исполняемым в данный момент процессам.

root — каталог системного администратора. Он находится в главном каталоге, т. к. в случае восстановления системы может не быть возможности для загрузки вспомогательных ФС или монтирования дополнительных устройств.

run — может содержать разную информацию, необходимую для функционирования ОС и процессов, например сокеты принадлежащий процессу.

sbin — системные утилиты, нужные для организации работы пользователя или используемые пользователем.

selinux — каталог в АльтЛинукс используется для специального программного обеспечения, часто пуст.

srv — чаще используется для работы прикладных служб.

sys — необходим для работы ОС.

tmp — содержит временные файлы. Его содержимое очищается после перезагрузки ОС.

usr — дублирует основные каталоги корневой ФС. Используется для хранения пользовательских программ, библиотек, настроек, игр, подключаемых файлов компилятора и прочей информации процессов пользователей. Такое дублирование необходимо с целью защиты ОС от неправильных действий пользователей.

var — используется для хранения различной информации, начиная от сокетов и уникальных идентификаторов процессов, заканчивая каталогами для

автомонтирования съемных накопителей. Предположительно автомонтирование производится в этот каталог, чтобы исключить совпадение имен при ручном монтировании устройств в каталогах media или mnt. Основное назначение информации этого каталога, хранение журналов исполнения различных сервисов и процессов. Здесь сохраняется информация о событиях, произошедших в ОС, переменные данные ОС. Рекомендуется размещать этот каталог на отдельном логическом или физическом накопителе, т. к. информация в каталоге обновляется очень часто и это может вызвать повышенный износ дорогого накопителя или фрагментации ФС. При использовании накопителей на основе жестких дисков это вызовет замедление работы и быстрый его износ. Самое важное что нужно знать об этом каталоге, — его переполнение приводит к останову всей системы.

В завершении краткого обзора базовых концепций и понятий ФС отметим, что основной перспективной концепций ФС является переход на безфайловое или не иерархическое хранение информации. В этом случае для доступа к информации пользователь должен знать только атрибуты информации, например, примерное время создания или изменения, тип искомой информации, объем информации. Переход на такой тип взаимодействия был анонсирован примерно в начале 2000-х годов и до сих пор не существует ни одной реализации, способной достойно заменить обычную иерархическую структуру информации.

Другим направлением развития ФС является её интеграция с системами контроля версий. Подобные системы позволяют отслеживать изменение состояния файлов и их структуры в процессе работы и позволяет возвращать ФС в состояние на любой момент времени.

1.7. Конфигурационная информация

Операционная система содержит в себе множество таблиц. Примером могут служить таблицы дескрипторов открытых файлов или сокетов, страничная организация оперативной памяти. Эти таблицы на прямую доступны только разработчикам ОС. Другой важной составляющей ОС является конфигурационная информация. Основной способ хранения конфигурационной информации использование конфигурационных файлов, содержимое которых в текстовом виде описывает настройки программного обеспечения. Этот подход имеет два способа применения. В одном случае конфигурационные файлы хранятся в специальном каталоге /etc . Преимущество такого подхода в том, что место поиска конфигурационного файла определено заранее. Другой вариант заключается в хранении конфигурационного файла вместе с исполняемым файлом.

Оба этих варианта имеют ряд недостатков и чтобы их преодолеть, компания MicrosoftTM разработала технологию специальной базы данных, называемой **реестр** WindowsTM. В этом случае все настройки хранятся в одной базе данных, а ОС предоставляет набор функций для добавления, изменения и удаления настроек. Для использования технологии реестра необходимо знание приоритетов настроек, описанных в его разных частях. По-факту реестр WindowsTM является одним из самых уязвимых мест ОС, т. к. технология реестра замыкает на себе все процессы, включая драйверы устройств и их взаимодействия в ОС независимо от их прав доступа. В результате элементы реестра ОС WindowsTM все равно хранятся в разных файлах, а сама компания MicrosoftTM со временем задумалась об обходе ограничений собственной

технологии, что было отражено в **платформе .NET**. Формат исполняемого файла в этой технологии позволяет хранить конфигурационную информацию непосредственно в исполняемом файле. Такой подход тоже противоречит основам разграничения доступа, т. к. исполняемые файлы обычно не должны иметь право на запись в самого себя, чтобы уменьшить возможный ущерб в случае возникновения ошибок при выполнении в оперативной памяти или заражении вирусом.

Большая часть конфигурационной информации в ОС АльтЛинукс храниться в каталоге /etc, расположенном в корневом каталоге. В нем находится как системная конфигурация ОС, так и настройки пользовательских программ. Общепринятой практикой стало использование в названии конфигурационного файла имени исполняемого файла программы, названия программы или сокращения, связанного с программой.

Для примера приведем программу acpi. Её назначение заключается в отображении информации из файловой системы /proc о состоянии заряда батареи, температуры, передаваемой датчиками в случае их наличия, прочей информации. Конфигурационная информации утилиты расположена в каталоге /etc/acpi. Этот каталог содержит вложенный каталог events. В ней имеется только один конфигурационный файл power, который является текстовым файлом в кодировке ASCII. Полный путь к нему /etc/acpi/events/power. Он имеет следующие права доступа: -rw-r--r-- . Его владельцем является пользователь root. Файл принадлежит группе root . Содержимое этого файла — две строки

event=button/power

action=/sbin/poweroff

Они предписывают исполнять программу /sbin/poweroff при нажатии кнопки power.

Процессы, запускаемые при загрузке ОС Линукс, имеют различные приоритеты запуска. Для конфигурирования и запуска таких процессов можно использовать каталог /etc/init.d/ , который является символьной ссылкой на каталог /etc/rc.d/init.d/ . Однако в ОС есть и другие уровни приоритетов запуска. Они располагаются в каталоге /etc/rc.d . Файлы в каждой из шести каталогов, задающих приоритет загрузки rc0.d — rc6.d, начинаются с символов K05. Число 05 соответствует желаемому порядковому номеру для запуска процесса в своей группе, задаваемой названием каталога, однако ОС не всегда может четко следовать этим предписаниям.

В каталоге /etc/ конфигурационные файлы могут быть в двоичном формате, например /etc/localtime . В нем записана временная зона и правила перехода на летнее/зимнее время.

Приведем другие примеры конфигурационной информации ОС.

- /etc/default/ — в этом каталоге хранятся файлы, используемые при создании новой учетной записи пользователя в системе.

- /etc/passwd — в этом файле хранятся пароли пользователей.

- /etc/group — в этом файле хранятся пароли пользователей.

- /etc/local/ - настройки текстового режима системы.

- /etc/kde/ — настройки графической системы KDE и её приложений.

- /etc/ppp/ - настройки демона (сервиса) pppd, реализующего PPP-интерфейс. Этот интерфейс используется для обеспечения работы модемов.

- /etc/NetworkManager — настройки менеджера сетевых подключений.

- /etc/samba/ - настройки демона, обеспечивающего работу в сети, содержащей компьютеры на базе Windows.

- /etc/sysconfig/ - каталог, содержащий файлы системной конфигурации.

- /etc/X11/ - настройки графической оболочки X11.

- /etc/hosts — содержит сопоставление IP адресов — символьным именам.

- /etc/motd/ — сообщение, выдаваемое системой после входа пользователя в систему.

- /etc/fstab — содержит список точек монтирования ФС, в том числе подключенные аппаратные накопители информации.

В отличие от настроек ОС, конфигурационные файлы пользователей могут находиться в его домашнем каталоге, например /home/ivanov/ . Обычно названия этих каталогов или файлов начинаются с символа точка, который предписывает не отображать такие объекты для пользователя без явного указания показывать скрытые объекты. Также конфигурационная информация пользователя может находится в каталоге /usr/etc/ .

В домашнем каталоге пользователя находится файл .dmrc, в котором указывается основной язык для сессии пользователя, например так:

```
[Desktop]
Language=ru_RU.utf8
Session=default
```

В каталоге .config располагается каталог autostart, в котором размещаются скрипты и объекты-иконки рабочего стола, выполняемые при входе пользователя в сеанс ОС.

Если нужный конфигурационный файл не удается найти в ФС, значит программа не установлена. Одним из способов поиска файлов конфигурации является изучение установочного скрипта в дистрибутиве.

1.8. Сетевые технологии. Стек протоколов TCP/IP

Одной из главных причин популярности вычислительной техники стало массовое распространение вычислительных сетей. Появление сети Интернет позволило кардинально снизить стоимость бытового общения на любые расстояния. Основные идеи обмена информацией с помощью вычислительных машин отражены в модели ВОС (OSI). **OSI** - open systems interconnection basic reference model. Базовая Эталонная Модель Взаимодействия Открытых Систем (**ЭМВОС**) или сокращенно **ВОС**. Модель описана в ряде стандартов:

1. ГОСТ Р ИСО/МЭК 7498-1-99. — «ВОС. Базовая эталонная модель. Часть 1. Базовая модель».
2. ГОСТ Р ИСО 7498-2-99. — «ВОС. Базовая эталонная модель. Часть 2. Архитектура защиты информации».
3. ГОСТ Р ИСО 7498-3-97. — «ВОС. Базовая эталонная модель. Часть 3. Присвоение имён и адресация».
4. ГОСТ Р ИСО/МЭК 7498-4-99. — «ВОС. Базовая эталонная модель. Часть 4. Основы административного управления».

Прикладной	HTTP, SMTP, SNMP, FTP, Telnet, NFS, RTCP
Представительный	XML, XDR, SMB
Сеансовый	TLS, SSH, RPC, NetBIOS
Транспортный	TCP, UDP, BGP
Сетевой	IP, ICMP, IGMP, X.25
Канальный	Ethernet, PPP, ISDN, ISDL
Физический	электричество, радио, лазер (физическая среда и принципы кодирования информации)

Полная модель ВОС называется традиционной моделью и содержит 7 уровней. Каждый протокол описан в RFC или другом документе.

В практической деятельности не всегда задействуются все 7 уровней стандартной модели ВОС. Например, при нажатии на ссылку в браузере может быть отправлен запрос по HTTP протоколу. В этом случае вычислительная машина отправителя запроса использует следующую схему формирования и передачи запроса:

Прикладной — HTTP.

Транспортный — TCP.

Сетевой — IP.

Канальный — Ethernet.

Физический — электричество.

Всего использовано 5 уровней вместо 7. Сервер получит сообщение и для его разбора задействует следующие уровни:

Физический — электричество.

Канальный — Ethernet.

Сетевой — IP.

Транспортный — TCP.

Прикладной — HTTP.

Таким образом на сервере также будут использованы протоколы 5 уровней.

Стеком TCP/IP называют набор протоколов разных уровней модели ВОС. Его главной особенностью является независимость от физической среды передачи информации. С помощью него реализуются все уровни модели ВОС. Сам он занимает 4 уровня: прикладной, транспортный, межсетевой, уровень доступа к сети (физический, канальный и частично сетевой). До сегодняшнего момента существуют разногласия на какие уровни делить стек протоколов. Например, предлагается ввести восьмой уровень «Internetworking» — между

транспортным и сетевым уровнями, т. к. существуют протоколы, которые нельзя отнести к одному из уровней.

Способ подключения вычислительных машин к компьютерной сети и архитектуру связей, сформированных между вычислительными машинами, называют **топологией** вычислительной сети. На сегодняшний день наиболее популярной является **топология звезда**, при которой к центральному серверу или маршрутизатору подключаются остальные вычислительные устройства. Топология сетей определяется канальным уровнем.

В стеке TCP/IP верхние 3 уровня модели ВОС (прикладной, представительный и сеансовый) объединены в прикладной, а функции по определению типа данных переданы программе пользователя.

В результате упрощенную модель ВОС для стека TCP/IP можно ограничить уровнями:

- физический — описывает среду передачи информации, принципы передачи информации в этой среде, физические характеристики этой среды;
- канальный — описывает, как выполняется кодирование и передача пакетов с информацией в используемой среде;
- сетевой — используется для передачи пакетов из одной подсети в другую, получения служебной информации, управления широковещательными и прочими пакетами;
- транспортный — используется для решения проблемы негарантированной доставки пакетов информации, гарантируют правильную последовательность получения пакетов, определяют источник и получателя пакета;
- прикладной — используется большинством сетевых приложений (браузеры, почтовые сервисы, передача

файлов и т.д.);

Существует набор стандартных сетевых протоколов, которые определяются Агентством по выделению имен и уникальных параметров протоколов (**IANA**). Основное отличие стандартных протоколов в максимально широкой поддержке вычислительными устройствами и присутствием зарезервированного номера порта, который закреплен только за этим протоколом. **Сетевой порт** — это участок оперативной памяти, в который копируются данные из памяти сетевого устройства.

В основе стека протоколов TCP/IP лежит протокол **IP** (Internet Protocol — межсетевой протокол). Он не обеспечивает надежную доставку, т. е. пакеты могут прийти не в том порядке, в котором были отправлены, прийти поврежденными или не прийти вообще. В современных сетях используется IPv4 (**RFC 791**) или более новый IPv6 (**RFC 2460**).

Основное понятие протокола IP — это IP пакет. **IP пакет (IP датаграмма)** — форматированный блок информации, передаваемый по вычислительной сети. Общепринятым способом описания структуры IP пакета является графическое представление датаграммы, которое представлено на изображении ниже. Представление датаграмм для всех протоколов нижних уровней включено в соответствующие им стандарты.

0				1								2								3							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Версия				IHL				Тип обслуживания								Длина пакета											
Идентификатор																Флаги				Смещение фрагмента							
Число переходов (TTL)								Протокол								Контрольная сумма заголовка											
IP-адрес отправителя (32 бита)																											
IP-адрес получателя (32 бита)																											
Параметры (до 320 бит)																Данные (до 65535 байт минус заголовок)											

Пояснения для всех полей даны в RFC 791. Интерпретация поля «Протокол» дана в **RFC 1700**, упоминание о котором многократно встречается в протоколах IP и 4-й и 6-й версий. Научится читать датаграммы очень просто. Приведенное графическое изображение подразумевает, что 1 и 2 строки содержат номера байт и битов, 3-я содержание первых 4-х байт. Путаница может возникнуть уже на этом этапе, т. к. первый байт имеет индекс 0. Четвертая строка содержит 5,6,7 и 8 байт IP пакета с номерами 4,5,6 и 7 соответственно. Для экономии ширины изображения, четвертая и последующие строки изображаются под третьей и без нумерации бит и байт, в которых они располагаются. Получается, что адрес получателя содержится в 17,18,19 и 20 байте IP пакета. Высота строки не указывает на количество байт в поле и зависит от размера текстового описания поля.

Датаграмма IP пакета версии 6 IPv6 имеет совершенно другой вид. Её заголовок занимает большее количество байт.

Версия (4 бита)	Класс трафика (8 бит)	Метка потока (20 бит)	
Длина полезной нагрузки (16 бит)		След. заголовок (8 бит)	Число переходов
IP-адрес отправителя (128 бит)			
IP-адрес получателя (128 бит)			
Данные			

В протоколе IPv6 учтена возможность определения приоритета пакета с помощью поля «Класс трафика». Его применение позволит ускорять передачу данных, актуальность которых быстро обесценивается со временем.

IP адреса распределены по области применения. В **RFC 1918**, а также в стандартах, указанных в его 9-м разделе, определены диапазоны адресов, которые могут

использоваться только в локальных сетях.

- 10.0.0.0 — 10.255.255.255 класс A;
- 172.16.0.0 — 172.31.255.255 класс B;
- 192.168.0.0 — 192.168.255.255 класс C;
- сеть 2001:0DB8::/32 в IPv6 — зарезервировано для примеров и документации.

Вычислительные машины объединяются в подсети общими начальными битами адреса. Количество таких бит называется маской подсети. **Классы IP адресов** определяют, сколько октетов адреса может изменяться. **Класс A** позволяет присваивать адресам\ вычислительным машинам, изменяя три октета, начиная справа. **Класс B** позволяет изменять два октета, начиная справа. **Класс C** только один октет, начиная справа. На данный момент классы не имеют повсеместного распространения в глобальных сетях, но могут применяться в отдельных локальных сетях.

Также существует специальный локальный адрес **127.0.0.1** . В случае отправки пакетов ему, они закольцовываются и передаются в порт этой же вычислительной машины. Данные, переданные адресу 127.0.0.1, не достигают сетевого устройства, соответственно он доступен даже при отсутствии или неисправности сетевого устройства. Благодаря этому возможна организация взаимодействия процессов в рамках одной ЭВМ даже при отсутствии сетевого утсройства.

Все современные технологии работают с использованием IP адресов. Не имея своего корректного IP адреса невозможно получить ответ на запрос, отправленный в вычислительную сеть по протоколу IP или более высокого уровня. Аппаратное обеспечение выполняет обмен информацией с помощью протоколов более низкого уровня и использует аппаратные адреса

(MAC адреса). Существуют технологии перехвата сетевого трафика, но о них речь пойдет в разделе, посвященном администрированию вычислительной сети. При подключении к точке доступа в вычислительную сеть, первой процедурой является автоматическое получение IP адреса по технологии **DHCP** или использование IP адреса, указанного заранее в ОС. Если IP адрес указывается вручную и ЭВМ с таким адресом уже присутствует в сети, передача данных производится не будет, а также будет нарушена стабильная работа ЭВМ с таким же адресом.

При построении глобальной сети Интернет, магистральных каналов, соединяющих материки и регионы, высокоскоростных оптоволоконных каналов, используются IP адреса, выделяемые организацией **ICANN**. Организация имеет свои представительства во всех крупных промышленно развитых регионах. В Европейском союзе представительство называется **RIPE NCC**. Без функционирования этих организаций создание глобальной вычислительной сети было бы невозможно, кроме того деятельность в сети Интернет регулируется международным правом и множеством локальных правовых баз. Считать Интернет безконтрольной зоной полной свободы не верно. Таким Интернет не был ни одного дня с момента своего создания.

Подключение ЭВМ с одинаковыми локальными IP адресами возможно. Для этого используются прокси-сервера или NAT. **Прокси-сервер** — это технология, предоставляющая для любой подключенной ЭВМ доступ в вычислительную сеть от имени прокси-сервера. Технология **NAT** предоставляет доступ ЭВМ с IP адресами своей локальной сети, в глобальную сеть. Принцип работы технологии NAT заключается в закреплении портов за отдельными локальными ЭВМ.

Для внешних ЭВМ подключения по NAT выглядят как обычные ЭВМ, с подключениями через один IP адрес.

На одном уровне с протоколом IP в модели ВОС находится протокол **ICMP RFC 792** с дополнениями **RFC 950** (Internet Control Message Protocol — межсетевой протокол управляющих сообщений). Как следует из названия протокол используется для решения сервисных задач: поиска ошибок, проверки доступности удаленной ЭВМ с помощью эхо-запросов. Несмотря на то, что этот протокол вынесен в отдельный документ, он является обязательным при реализации стека TCP/IP. Датаграммы протокола ICMP фактически помещаются как пользовательские данные в датаграммы протокола IP, однако данные, пересылаемые в нем, поддерживают работу протокола IP и их обработка включена в алгоритмы протокола IP. У каждой версии протокола IP есть соответствующая версия протокола ICMP этой же версии, IPv4 — ICMPv4, IPv6 — ICMPv6.

Приведем пример датаграммы протокола ICMPv4.

Бит	0 — 7	8 — 15	16 — 31
0	Тип	Код	Контрольная сумма
32	Содержание сообщения(зависит от значений полей Код и Тип)		

Версия используемого протокола не указывается, так как соответствует версии протокола IP, которая, в свою очередь, указывается в первых четырех битах заголовка IP пакета.

Два основных протокола стека протоколов TCP/IP — **TCP** (**RFC 793**) и **UDP** (**RFC 768**). Протокол UDP служит для негарантированной доставки данных. Негарантированная доставка означает, что данные могут не придти к адресату, могут придти не полностью или в неправильном порядке. Он используется при передаче потоковой мультимедийной информации и в других

случаях, когда актуальность информации снижается каждую долю секунды. Протокол TCP наоборот служит для гарантированной доставки, т. е. в каком виде данные были отправлены в сеть, в таком они будут получены адресатом. Необходимо обратить особое внимание! Порты протоколов TCP и UDP с **одинаковыми** номерами — это **разные** участки оперативной памяти. Протокол UDP работает быстрее и в локальных вычислительных сетях потери данных почти не происходит, однако, если необходимо гарантировано доставить некоторую информацию, нужно использовать более медленный протокол TCP. Существует **золотое правило** выбора между протоколами TCP и UDP, если хотя бы раз потребовалось подтвердить получение данных, необходимо использовать протокол TCP. Реализация любой надстройки над UDP с целью подтверждения получения данных всегда будет заведомо проигрышное решение. Проблемы могут возникнуть как на этапе разработки, так и на этапе поддержки или при выпуске новой версии через несколько лет эксплуатации программного обеспечения.

Максимальная длина UDP пакета составляет 65527 октетов (байт). Для взаимодействия сетевых приложений используются 16-ти битные порты со значениями от 0 до 65535. Порт с номером 0 считается портом источника и используется, если программа не требует ответных действий. Порты с 1 по 1023 системные. Их назначение не может быть изменено, доступ к ним возможен только с привилегиями администратора системы. Порты с 1024 по 49151 — относятся к зарегистрированным портам, которые используются специальным программным обеспечением. Порты с 49152 по 65535 — используются для пользовательских приложений. Для отправки данных по UDP не требуется становления соединения. UDP

используется во множестве протоколов, наиболее популярными из которых являются: DNS, RTP, RTCP, SNTP, NTP.

Протокол TCP предоставляет поток данных, с предварительным установлением соединения. В случае потери данных TCP самостоятельно производит их повторный запрос, отбрасывает повторные данные, исправляет множество других коллизий. Если происходит разрыв соединения на длительное время, соединение сбрасывается и выдается сообщение об ошибке. Протокол TCP используется множеством протоколов: HTTP, FTP, RTCP, NFS, Torrent и другими.

Адрес отправителя и получателя содержится в заголовке IP датаграммы. В заголовке TCP датаграммы, как и в заголовке UDP датаграммы, содержится порт отправления и порт назначения.

В отличие от рассмотренных ранее протоколов, в которых данные передаются в двоичном виде, в протоколе HTTP (HyperText Transfer Protocol — «протокол передачи гипертекста») данные передаются в текстовом виде и имеют значительную избыточность. Другим отличием протокола HTTP является его базирование на архитектуре «клиент — сервер». При его использовании взаимодействие может быть осуществлено только по запросу «клиента». «Сервер» обрабатывает запрос, формирует ответ и соединение разрывается. Адрес ресурса задается с помощью **URL** в запросе клиента. Наиболее популярны следующие версии этого протокола: HTTP 1.0 (**RFC 1945**), HTTP 1.1 (**RFC 2616**), HTTP 2.0 (**RFC 7231**).

В сетевых технологиях активно используются относительные понятия **локальный** и **глобальный**. Локальной может являться вычислительная сеть организации, но она является глобальной для

вычислительной сети одного отдела. Также локальной может быть вычислительная сеть страны, по отношению к Интернет. Однако вычислительная сеть страны одновременно является глобальной для сети одного региона.

Перечислим классы архитектур серверов, наиболее популярные в сети Интернет. По способу организации памяти они делятся на однопоточные и многопоточные. В случае **однопоточного** сервера производится последовательная обработка запросов, поступающих на сервер. Остальные запросы ставятся в очередь на обработку или теряются. **Многопоточный** сервер создает для каждого входящего соединения отдельный поток, который выполняет его обработку, формирует ответ, возвращает ответ клиенту и завершает свою работу. Другое деление архитектур связано с направлением передачи данных. **Асинхронный** сервер может обмениваться данными с клиентом в один момент времени только в одном направлении, т. е. только от сервера к клиенту, или от клиента к серверу. **Синхронный** сервер может осуществлять двунаправленную передачу данных в любой момент времени.

Для глубокого изучение межсетевых взаимодействий и стека протоколов TCP/IP требуется не менее года ежедневной работы. В институте Беркли авторский курс в этой области обеспечен материалом, изложенным в трех томах. Этот раздел необходим для обозначения основных технологий и протоколов, которые будут использоваться далее и не являются учебным пособием.

Отдельно следует рассматривать семейство сетевых протоколов, в которые изначально включены средства защиты передаваемых данных, например, протоколы IPsec, HTTPS, HFTP.

2. Администрирование ЭВМ, вычислительных сетей

2.1. Установка и обновление ОС и ПО

В этом разделе описаны основные этапы подготовки программного обеспечения к разработке программного обеспечения. Главу правильнее было бы назвать «Обслуживание ЭВМ и компьютерных сетей», потому что в ней будут рассмотрены всего несколько задач администрирования.

В разделе нет подробных пошаговых инструкций, которые имеются в различных формах в сети Интернет. Каждый человек имеет свои предпочтения при знакомстве с новыми знаниями, поэтому здесь поясняются исключительно вопросы, которые сложно сформулировать и еще сложнее найти на них правильные ответы. Также в этом разделе не затрагиваются вопросы выбора аппаратного обеспечения, считая, что перед покупкой пользователь догадался найти информацию о совместимости выбранного аппаратного обеспечения с другими компонентами и о его поддержке современными операционными системами.

До работы с ЭВМ необходимо пройти этап подготовки. Первым этапом подготовки является определение набора задач, для решения которых готовиться ЭВМ. ПЭВМ на базе ядра ОС Линукс может решать все задачи, за исключением эффективного использования программного обеспечения, разработанного специально для ОС WindowsTM. После определения задач, необходимо заняться сбором информации об удобстве их решения в разных дистрибутивах.

Для первоначального сбора информации достаточно уметь пользоваться поисковыми машинами в сети

Интернет. На сегодняшний день текстовые материалы лучше использовать для аналитического сравнения ОС, программ, способов реализации функций. Видеоматериал лучше использовать для подготовки к некоторому действию, например, установке ОС.

После определения ряда дистрибутивов, которые позволяют решать все поставленные задачи, выбор конечного можно производить, руководствуясь самым главным критерием: простота эксплуатации и обслуживания, наличие необходимого программного обеспечения. На практике у начинающих пользователей, выбор часто производится по причине наиболее красивого изображения на рабочем столе, что тоже очень важно.

Существует еще одна не формализуемая причина выбора дистрибутива: «так сложилось исторически». Обычно под этим подразумевают наличие значительного опыта его применения. В текущей ситуации такое обоснование является подтверждением того, что над этим дистрибутивом работает живое сообщество разработчиков и пользователей, также имеется бесплатная оперативная поддержка в виде форума по адресу http://www.forum.altlinux.org.

Для получения дистрибутива следует выбирать исключительно доверенные источники. Чаще всего это сайт в сети Интернет разработчика дистрибутива. В случае возникновения необходимости использовать файл-образ ОС из не доверенного источника, например, из сети, работающей по протоколу torrent, необходимо подтвердить его подлинность с помощью контрольной суммы с сайта разработчика. Как это сделать будет описано в следующих разделах.

Разработчики АльтЛинукс предоставляют две концепции распространения дистрибутивов. На сайте

разработчиков по адресу http://www.altlinux.ru имеются ссылки на актуальный дистрибутив АльтЛинукс. Это полноценный дистрибутив с репозиторием программ для него содержащим порядка сотни тысяч бесплатных программ.

Для высококвалифицированных пользователей, администраторов и разработчиков предоставляется дистрибутив в виде стартовых наборов. На сайте разработчиков по специальному адресу https://www.altlinux.org/Starterkits имеется информация о них. Стартовые наборы для скачивания доступны по адресу http://nightly.altlinux.org/ . Данные наборы содержат в себе минимальное количество пакетов программ.

Далее для описания будет использоваться дистрибутив AltLinux. На сайте разработчиков присутствуют образы двух форматов: ISO и IMG.

Для того чтобы выполнить установку ОС на ЭВМ, необходимо сделать загрузочный CD/DVD диск или загрузочный флеш-накопитель. Для создания CD/DVD используется файл-образ формата ISO, для создания загрузочного флеш-накопителя — файл-образ формата IMG. Эти форматы называются файл-образами, потому что кроме файлов, необходимых для загрузки ОС, они содержат информацию о их расположении на накопителе, на котором они использовались в качестве ОС.

Для записи файлов-образов в ОС WindowsTM используются специальные программы. В ОС, основанных на ядре Линукс достаточно выполнить следующие действия:

1. скачать файл-образ из сети Интернет, например, с именем cdrom.img ;

2. переместить его в папку /mnt/img/ ;

3. определить путь к устройству, записывающему CD, например /dev/sr0 ;

4. выполнить команду: dd if=/mnt/img/cdrom.img of=/dev/sr0 .

При наличии у пользователя опыта работы с виртуальными машинами возможна установка ОС Линукс в качестве виртуальной машины, независимо от используемой системы виртуализации virtualbox, wmware, kvm (qemu). Также как и с другими ОС, при использовании Линукс на виртуальной машине происходит падение производительности центрального процессора, также существенно снижается производительность видеокарты. Некоторые аппаратные и программные мультимедия технологии, драйвера разработчика видеокарты могут быть недоступны вообще. К преимуществам использования виртуальных машин следует отнести меньший риск для базовой ОС, возможность эмуляции отсутствующего аппаратного обеспечения, например, 8 сетевых карт, вместо имеющейся одной.

ЭВМ и ПО, как любое техническое решение или его часть, требует обслуживания. Одна из главных задач обслуживания ПО — его обновление. В ОС WindowsTM используется общая стратегия, говорящая о том, что все обновления должны быть установлены как можно быстрее. Принимать решение о нужности того или иного обновления рекомендуется только опытным пользователям. Большие обновления происходят примерно один раз в месяц, так называемые сервисные пакеты могут выходить примерно один раз в год. При этом речь идет только об обновлениях самой ОС, но не о стороннем ПО. В каждом дистрибутиве ОС Линукс существует своя стратегия обновления. Разработчики АльтЛинукс рекомендуют выполнять обновление по

необходимости. Финальные сертифицированные дистрибутивы выпускаются компанией раз в год или несколько лет. Это считается достаточным. Исключение существует для исправлений обнаруженных критических уязвимостей безопасности и критических ошибок, приводящих к зависанию программ. При этом имеется ввиду все программное обеспечение вместе с ОС.

В сообществе АльтЛинукс обновления появляются значительно чаще, практически ежедневно. Они используются для тестирования и не обязательны для установки. Даже в сообществе рекомендуется выполнять обновления не чаще, чем раз в квартал или по необходимости.

Перед обновлением в АльЛинукс, необходимо запустить программу **Synaptic** и проверить настройки репозиториев (Главное меню → Параметры → Репозитории). **Репозиторий** – каталог на сервере, имеющий специальную структуру, в котором хранится программное обеспечение и ядро операционной системы в виде исходных файлов или пакетов для конкретной аппаратной платформы ЭВМ. Наиболее популярными платформами на текущий момент являются (x86, x86-64). При первом запуске необходимо удостовериться, что в настройках репозитория выбран хотя бы один репозиторий, адрес которого начинается с названия протокола **http**, **ftp** или **rsync**, т.е. напротив строки выставлен символ «**V**» в элементе множественного выделения.

ПО хранится на сервере в виде пакетов формата **rpm** или **deb**. Пакеты являются архивированными каталогами с файлами. Принципиальным отличием этих форматов является концепция. Пакеты формата rpm обычно содержат только устанавливаемое ПО и список зависимостей от других пакетов, т.е. список программ и

их версий, без которых эта программа работать не может. Пакеты формата deb обычно содержат все необходимые для установки и запуска программы. Наиболее ярко преимущество концепции deb пакетов проявлялось в период высокой стоимости доступа к сети Интернет и в плохо развитых системах хранилищ rpm пакетов. Основной недостаток deb пакетов — в возможности захламления накопителя с ОС различными версиями одной программы при частой установке и удалении программ. Сейчас эти концепции одинаково популярны. В АльтЛинукс разработчиками используются rpm пакеты, однако возможно использование и deb пакетов, но это требует углубленных знаний по администрированию ОС Линукс.

Пакетами в формате rpm управляет программа **apt**. С помощью неё проверяются требуемые зависимости, выполняется обновление списков доступных программ, устанавливаются и удаляются программы. Synaptic является графическим интерфейсом для apt, поэтому все изменения, выполняемые в Synaptic, можно сделать путем внесения изменений в конфигурационные файлы apt. Следует помнить, что apt базируется на собственном хранилище данных и не является системой управления базами данных. В связи с этим одновременно может быть запущен только один экземпляр программы apt. Пользователи с небольшим опытом работы могут попробовать параллельно запустить apt и Synaptic, что вызовет блокировку программы, запущенной второй, т.к. они обе обращаются к одному хранилищу данных.

Программа apt конфигурируется с помощью каталога apt в каталоге /etc/ . В этом каталоге содержится файл sources.list, в котором указывается путь к файлу, содержащему адреса репозиториев. В текущей версии ОС пути находятся в файлах каталога

/etc/apt/sources.list.d/*.list . Такая путаница возникла в процессе развития программы apt и необходимости разделить все имеющиеся пути к репозиториям, например, по имени компании, предоставляющей к ним доступ.

Компания — разработчик дистрибутива АльтЛинукс делает снимки репозиториев по датам и размещает их в отдельных каталогах. Такой подход позволяет реализовывать следующий прием. В случае, если последнее обновление привело к отказу ОС, требуется наиболее новая версия одной из программ, необходимо в качестве репозитория указать такой снимок. Например:

```
rpm
http://ftp.altlinux.org/pub/distributions/archive/p8/date/2016
/09/15 x86_64 classic
rpm
http://ftp.altlinux.org/pub/distributions/archive/p8/date/2016
/09/15 noarch classic .
```

Снимок можно отличить от обычного репозитория по наличию в нем даты, когда он был создан.

В заключении остановлюсь на проблеме разграничения доступа, которую можно решить на этапе установки ОС. Первоначально под пользователем ОС подразумевался человек. Сейчас такое соответствие не обязательно и практически никогда не соблюдается. Пользователь и соответствующая группа может создаваться под отдельное устройство или программу. В связи с этим, если задачи, которые планируется решать на ЭВМ, можно сгруппировать и эти группы не пересекаются, то под каждую из этих групп желательно создать отдельного пользователя. Решаемые задачи не обязательно должны быть уникальными, они могут повторяться.

Например:

- пользователь для игр;
- пользователь для работы в офисных приложениях прослушивания музыки и работы в сети Интернет;
- отдельный пользователь для прослушивания музыки и Интернет-серфинга;
- пользователь для тестирования новых программ и обновлений.

Такое деление позволяет, в случае некорректной работы одной из программ, удалить пользователя со всеми файлами, принадлежащими ему одной командой: userdel -r username . Настройки и файлы остальных пользователей не пострадают.

Другой обширной темой, изучение которой желательно проводить до установки Линукс, правильная разметка носителя информации, на который устанавливается ОС, слишком обширна для рассмотрения в контексте данного раздела. Также рекомендуется рассмотреть вопросы создания виртуальных файловых систем, но этот вопрос не относится к первостепенным.

2.2. Программное обеспечение

Ранее было сказано о том, что при выборе ОС необходимо определить свои потребности, сделав список задач, которые планируется решать с помощью данного вычислительного устройства. Под решением задачи имелось в виду наличие ПО, имеющего соответствующие функции. В этом разделе будет перечислено бесплатное программное обеспечение, которое автор использует на постоянной основе. В скобках после названия программ перечислены аналоги, которые используются в случае невозможности установки рекомендуемой программы.

Следует отметить, рабочая среда, в том числе рабочий стол, формируется платформой xfce4. Перечислим другие **пользовательские программы**:

- firefox, opera, chromium – программы-браузеры для просмотра документов в формате html;
- wget – программа для загрузки файлов из сети в автономном режиме, т.е. без вмешательства пользователя.
- libreoffice – полноценный оффисный пакет, поддерживающий как форматы файлов свободного распространения, так и закрытые форматы;
- qmmp – аналог популярного прогрывателя winamp для POSIX систем;
- qmmp-out-pulseaudio – плагин программы qmmp. После его установки его необходимо выбрать в свойствах qmmp. Плагин работает в подсистемой вывода аудио pulseaudio, которая установлена на вычислительной машине автора по умолчанию;
- sane – программа для использования устройства сканирования изображений с листа бумаги;
- xsane – графическая оболочка для программы sane;
- cuneiform – программа для распознования текста в

отсканированных файлах и файлах графического формата;

- okular (FoxIt Reader, CoolReader) – просмотрщик файлов в нескольких десятках форматов. В основном используется для просмотре файлов в формате pdf;

- fbreader – программа просмотра фалов формата электронных книг fb2;

- ristretto – простая программа из окружения xfce4 для просмотра графических файлов;

- kopete – программа для обмена сообщениями и файлами с поддержкой множества протоколов, включая XMPP и ICQ;

- VLC, kaffeine – проигрыватели мультимедийных файлов. Наличие двух программ связано с дублированием на случай неудачного обновления одного из них;

- XVidCap — позволяет производить запись видео действий, производимых на рабочем столе;

- K3b — программа с графическим интерфейсом. Выполняет создание образов форматов ISO и их запись на оптические носители информации;

- genisoimage — программа с терминальным интерфейсом, выполняющая создание образов файлов и каталогов в различных форматах;

- wodim — программа с терминальным интерфейсом, позволяющая производить запись оптических носителей (CD/DVD);

- wine — альтернативная реализация технологии WinAPI компании MicrosoftTM. Позволяет запускать простые приложения, реализованные с использованием этой технологии;

- synaptic — в AltLinux установлен по умолчанию. В других дистрибутивах также может устанавливаться как менеджер пакетов формата rpm, т. е. для установки

других программ;

- lyx – программа для работы с научным форматом подготовки документов TEX;

- wxmaxima – графическая оболочка для программы maxima, умеющего выполнять аналитические математические преобразования;

- blender — наиболее функциональный свободный редактор трехмерной графики, позволяющий создавать мультипликацию, видео и игры;

- inkscape — редактор векторной графики в формате SVG;

- gimp — наиболее функциональный свободный редактор растровой графики;

- ImageMagick-tools — пакет программ, позволяющий выполнять пакетную обработку изображений, а также редактировать изображения с помощью команд эмулятора терминала;

- audacity – наиболее функциональный свободный редактор аудио;

- kdenlive – один из наиболее функциональных свободных редакторов видео;

- avconv – набор инструментов для обработки видео с помощью команд в эмуляторе терминала;

- galculator – калькулятор, имеющий три режима работы (простой, инженерный, бумажный – формулы для вычислений набираются текстом);

- mc – менеджер файлов со встроенным текстовым редактором, возможностью подключения по ftp и множеством других функций;

- medit – графический текстовый редактор, удобный для редактирования текстовых файлов в различной кодировке;

- xscreensaver – заставки рабочего стола. Для настройки в эмуляторе терминала необходимо выполнить

команду xscreensaver-demo;

- rss-glx – заставки рабочего стола с использованием opengl.

Следующий список содержит **вспомогательные пользовательские программы**, установка которых также крайне желательна:

- java-1.8.0 openjdk. Пакет, в первую очередь, необходим для электронных таблиц и прочих программ пакета LibreOffice;
- rsync – программа для синхронизации файлов и каталогов;
- md5sum – программа для создания хеш-образов по стандарту md5;
- espeak – программа воспроизводит текст мужским голосом. Возможно написание текста на разных языках;
- gpg – криптопакет, программа для шифрования почтовых сообщений;
- enca – программа для преобразования текстовых файлов в различные кодировки;
- unrar, zip, p7zip – программы-архиваторы;
- bluez-utils – пакет программ для работы с радиоприемником, использующим протоколы семейства bluetooth;
- wireless-tools – пакет программ для работы с радиоприемником, использующим протоколы семества WIFI;
- gostcrypt – инструмент для шифрования по стандарту ГОСТ 28147-89;
- acpi — программа отображает информацию из файловой системы /proc о заряде батареи или с датчиков температуры;
- foomatic, hplib, cups – программы, необходимые для использования принтеров различных моделей.

Список далее содержит программное обеспечение, необходимое **для обслуживания** вычислительной машины:

- gparted – программа с графическим интерфейсом для разметки носителей информации;
- fdisk – программа для разметки носителей информации с терминальным интерфейсом;
- lsblk – программа выводит список разделов на носителях информации, точки монтирования, раздел с которого произведена загрузка ОС;
- hddtemp – программа отображает температуру накопителей, если они снабжены технологией S.M.A.R.T.;
- smartctl – отображает информацию S.M.A.R.T.;
- testdisk – программа восстановления информации на носителях информации, удаленной из файловой системы, но до физической перезаписи ячеек памяти носителя информации;
- gsmartcontrol, smartmontools – пакеты, содержащие программы smartctl, smartd;
- lm_sensors – пакет ПО, предоставляющий доступ к информации датчиков вычислительной машины. После установки может требовать выполнения команды sensors-detect. Для отображения информации датчиков используется програма sensors;
- aticonfig/nvidia-setting – набор драйверов и утилиты для конфигурирования видеокарт производителей ATI/NVIDIA;
- encfs – пакет для создания шифрованных каталогов;
- cryptsetup – программа для создания шифрованных разделов;
- htop – наглядный и функциональный аналог стандартной программы для отображения процессов и загрузки центрального процессора и оперативной

памяти;

- tcpdump – перехват и сбор данных по сетевым протоколам;
- flashrom – программа для создания образов и перезаписи микросхем вычислительного устройства;
- dmidecode – программа отображения информации BIOS.

Следующий список содержит программы, используемые **для разработки** программного обеспечения:

- dosbox – эмулятор ОС DOS TM (MicrosoftTM);
- libtool – набор инструментов для создания, установки и удаления библиотек;
- git – система контроля версий;
- curl – программа для отправки запросов по протоколу HTPP/HTTPS и получения ответов;
- sqlite3 – библиотека, встраиваемая в разрабатываемое ПО с целью реализации функций БД, на основе языка структурированных запросов SQL;
- geany – среда для работы с исходными кодами программ, с функциями автоматизации разработки ПО;
- valgrind – программа для поиска утечек оперативной памяти в период исполнения разработанной программы. Под утечкой подразумевается наличие ошибок, связанных с успешным выделением блоков оперативной памяти и не освобождением их после использования;
- meld – графический редактор, позволяющий сравнивать текстовые файлы и каталоги с текстовыми файлами. Показывает посимвольные отличия между файлами.
- gcc – компилятор языка программирования C;
- gcc-c++ - компилятор языка программирования C+

+;

- gdb – отладчик программного обеспечения, дополненного отладочной информацией на этапе компиляции;

- make – программа создания скриптов для компиляции сложных проектов;

- automake – программа конфигурирования скриптов компиляции сложных проектов;

- doxygen – программа автоматического создания документации из исходного кода программы. Для лучшего качества документирования требует создания специальных комментариев в коде программы;

- doxygen-wizard – программа-помощник doxygen;

- libusb-dev – набор для разработки библиотеки, взаимодействующей с устройствами по USB протоколу;

- libreadline-devel – набор для разработки библиотеки обработки текста в эмуляторе терминала;

- libudev-devel – набор для разработки библиотек, получающих информацию об устройствах;

- mariadb (mysql) – СУБД для малых и средних проектов реляционных БД;

- mingw – набор библиотек для компиляции для платформы WIN32;

- swi-prolog – свободная реализация языка Prolog.

Далее приведен список программ, используемых **в сфере информационной безопасности**:

- kismet – программное обеспечение для определения надежности радиосоединений WIFI. Позволяет выполнять поиск, перехват и вмешательство в функционирование соединений WIFI;

- wireshark – сетевой анализатор различных протоколов с графическим интерфейсом;

- tshark – сетевой анализатор с терминальным

интерфейсом;

- nmap – сканер сетевых портов;
- snort – автоматический анализатор и блокировщик сетевых пакетов;
- p0f – программа позволяет деанонимизировать версию ОС, на которой было сформировано сообщение;
- whois – программа для работы с протоколом WHOIS (**RFC 3912**). Позволяет определять владельцев сетевых ресурсов, их сетевые адреса;
- tor – сеть с «луковичным» шифрованием соединений. Позволяет оставаться анонимным в сети Интернет при квалифицированном использовании. Её применение может быть запрещено на территории некоторых стран;
- traceroute – программа, определяющая промежуточные узлы вычислительной сети и их сетевые адреса между источником и получателем информации;
- netstat – программа, отображающая все сетевые соединения, как локальные, так и связывающие с другими узлами вычислительной сети;
- netcat – программа отправки и получения сообщений по TCP и UDP протоколам.

Приведенное деление программ очень условно. Все это программное обеспечение можно на законных основаниях БЕСПЛАТНО устанавливать на своих вычислительных машинах без каких-либо ограничений. В случае установки платных аналогов, стоимость программного обеспечения для автора было бы в 40 и более раз больше стоимости самой вычислительной машины.

2.3. Эмулятор терминала

При запуске вычислительной машины производится проверка работоспособности её основных компонентов. Далее загружается загрузчик операционной системы, например **GNU Grub**, и отображается его графический интерфейс. С его помощью возможен выбор ОС, если установлено несколько ОС на одну ЭВМ, или выбор режима имеющейся ОС. Один из режимов загрузки **-single**, не выполняет загрузку графического интерфейса и производит загрузку одного суперпользователя **root**. В этом режиме загрузка производится в компьютерный терминал.

Базовым для понятия компьютерный терминал является **текстовый пользовательский интерфейс** (ТПИ). Этот интерфейс является разновидностью интерфейса пользователя и предоставляет возможность отображения информации и ввода/вывода информации с помощью буквенных, цифровых символов и символов псевдографики. ТПИ содержит в себе подкласс, называемый интерфейсом командной строки.

Интерфейс командной строки – текстовый интерфейс, имитирующий на экране прокручивающуюся бесконечную ленту бумаги, на которую пользователь может выводить текст вводимых им команд и следом за ними получать результаты их работы.

Интерфейс командной строки всегда используется в программном обеспечении содержащим сотни и тысячи команд. Такое количество команд просто невозможно разместить в графическом интерфейсе.

Другая проблема, решаемая интерфейсом командной строки, это возможность управлением вычислительными машинами с низкопроизводительным микропроцессором, каналом связи или при отсутствии

низкоквалифицированых пользователей. Таким требованиям удовлетворяют: датчики, встраиваемые устройства, мобильные устройства, составные компоненты персональных вычислительных машин, системы промышленной автоматизации, высокопроизводительные сервера без аппаратного интерфейса VGA. Отметим, что серверное оборудование с VGA интерфейсом массово поставляется только в одну страну мира – РФ.

Интерфейс командной строки следует из компонентной архитектуры. С помощью дополняющего его **пакетного интерфейса**, смысл которого в возможности создания текстовых файлов с перечислением команд, которые можно исполнить так, как будто их набрал пользователь, он позволяет выполнять автоматизацию действий.

Кроме того, интерфейс командной строки более информативный, из-за отсутствия постоянного отображения элементов самого интерфейса. Работу в нем проще обсуждать в сообществе, обмениваясь текстовыми сообщениями с использованием сети Интернет.

Основные недостатки интерфейсов данного вида – высокие требования к квалификации пользователя и отсутствие возможности управлять вычислительной машиной путем незначительных изменений положения своего тела.

Компьютерный терминал – устройство или набор устройств, используемые для взаимодействия оператора или пользователя с вычислительной системой локально или удаленно. К одной вычислительной машине может быть подключено множество терминалов. Например, при закрузке ОС АльтЛинукс, по умолчанию запускается сразу 4 интерфейса коммандной строки, переключение между которыми возможно с помощью комбинации

клавиш CTRL+ALT+F1, CTRL+ALT+F2. Для возврата в графический режим необходимо вернуться в интерфейс коммандной строки 1 или 7 в зависимости от настроек ОС. Не путать переключение между интерфейсами командной строки и переключениями между графическими рабочими столами выполняемое с помощью комбинаций CTRL+F1, CTRL+2 и т.д.

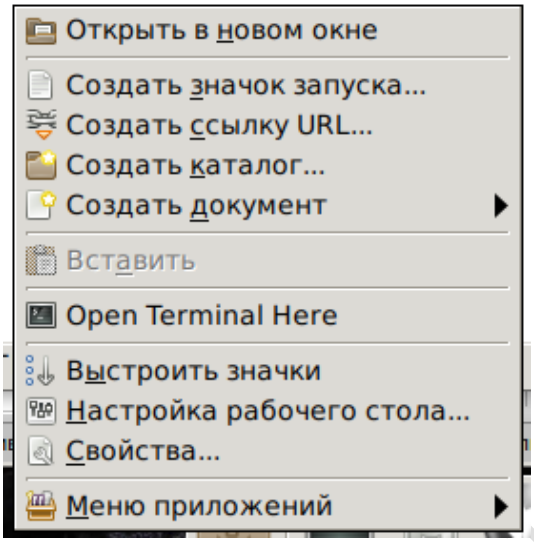
Компьютерный терминал может реализовывать любые интерфейсы, однако первые компьютерные терминалы могли содержать только интерфейс командной строки. Из-за этого на сегодняшний день укоренилось восприятие приведенных выше понятий как синонимом, что не верно.

Программа, эмулирующая компьютерный терминал внутри другой архитектуры или другого интерфейса, называется **эмулятором терминала (ЭТ)**.

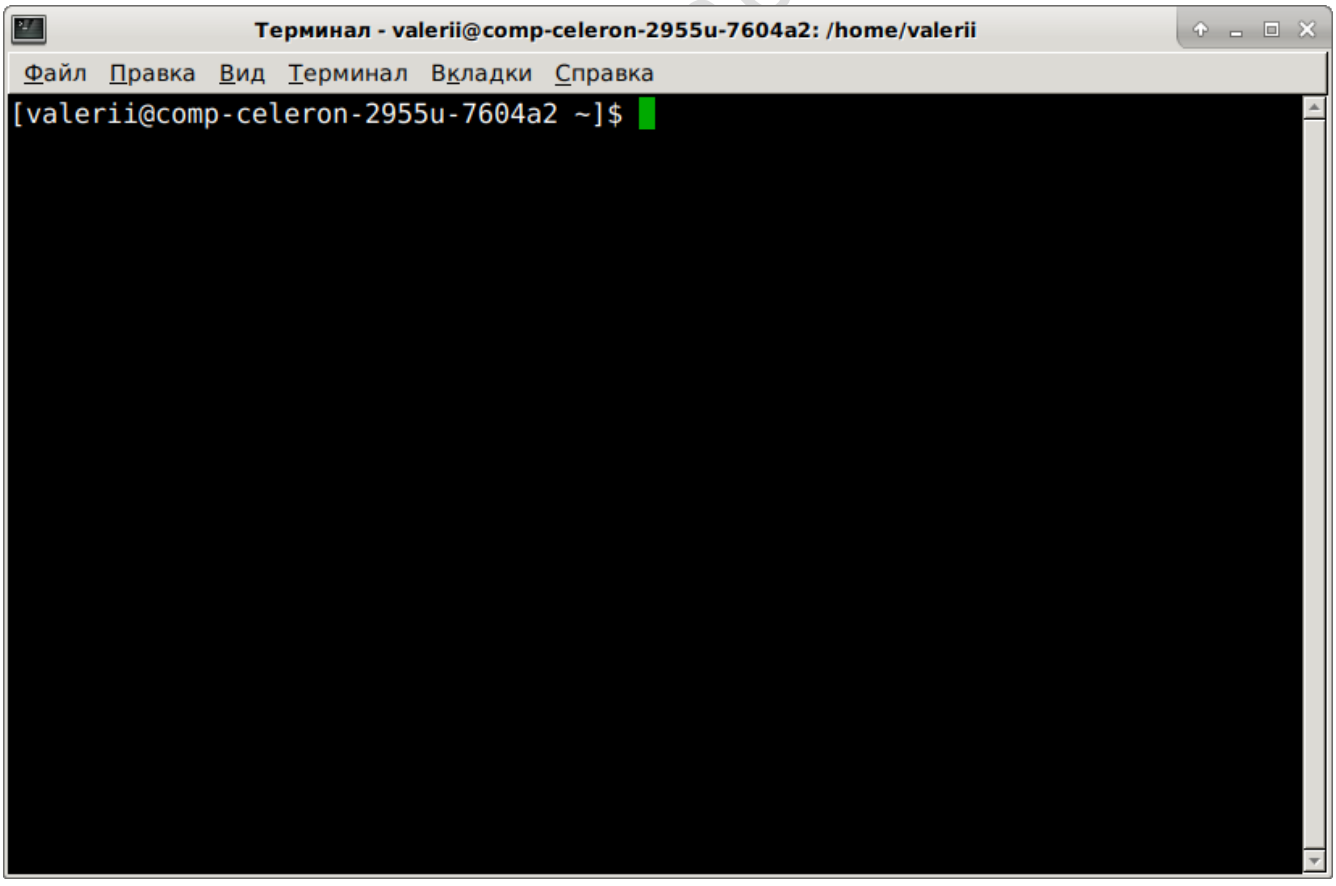
Использование **командного интерпретатора** в эмуляторе терминала позволяет выполнять автоматизацию действий при вводе любых команд. Наиболее популярным являются версии GNU Bourne-Again SHell, **bash** и **sh**, которые немного отличаются синтаксисом написания выражений.

Наиболее часто используемыми в практической деятельности являются эмулятор терминала и командный интерпретатор, остальные понятия приведены для понимания разницы между ними.

После загрузки ОС AltLinux с любой графической платформой и аутентификации пользователя отображается «**Рабочий стол**» пользователя. Вызвать терминал пользователя можно различными способами, например, с помощью пункта **Эмулятор терминала** в **Главном меню**. Также можно выполнить щелчок правой кнопкой мыши на любом свободном участке рабочего стола.



Далее необходимо выбрать пункт меню «**Open Terminal Here**» или «**Эмулятор терминала**». Откроется следующее окно.



Зеленый прямоугольник информирует о том, что фокус ввода передан эмулятору терминала. При вводе символа, символ будет отображаться на месте этого прямоугольника. Если фокус ввода передан в другое окно, прямоугольник станет черным, окаймленным зеленой рамкой. В каждой строке ввода команды отображается имя пользователя, отделенное символом амперсанд @. Далее следует сетевое имя вычислительной машины, пробел и имя текущего каталога. Для отображения имени домашнего каталога пользователя используется символ тильда ~.

Введите команду **ls /etc** и нажмите клавишу **Enter**. Символы ls — это имя программы, расположенной в каталоге /bin, которая отображает содержимое каталогов. Её имя является сокращением от английского list – список. Далее следует пробел и полный путь к папке etc, которая находится в корневом каталоге. Точку в конце команды ставить нельзя.

Каталог ls содержит множество объектов, поэтому после выполнения команды отобразится конец списка объектов каталога etc. Для перемещения вверх по отображенному списку используются комбинации клавиш **Shift+PageUp**, для перемещения вниз **Shift+PageDown**.

В любой момент начните вводить команду **exit** . Отображение в эмуляторе терминала будет прокручено вниз к области ввода команды. В дальнейшем ввод и просмотр отображения вывода в эмуляторе терминала стоит осуществлять таким образом.

В эмуляторе терминала есть две программы **info** и **man**. Это два формата справочной информации о других программах. Более популярной является система man, более удобной для пользователя info. Чтобы получить справочную информацию о команде ls, необходимо

выполнить команду

man ls

Рассмотрим наиболее часто используемые команды:

- **pwd** – вывод полного пути к текущему каталогу,
- **ls** – вывод списка объектов текущего каталога,
- **cd** – сменить текущий каталог,
- **echo** – отображение строк текста,
- **mkdir** – создание нового каталога,
- **rmdir** – удаление каталога,
- **touch** – изменение метки файла, создание файла без содержимого в случае его отсутствия,
- **rm** – удаление файла,
- **cat** – отображение содержимого файла,
- **chmod** – смена режима доступа к существующему объекту ФС,
- **chown** – смена владельца указанного объекта.

Приведем решение следующей **задачи**: необходимо в домашнем каталоге создать каталог с именем 123, в нем файл z.txt, в который записать текст «Hello» и на следующую строку символы «C!».

1. После запуска программы в эмулятора терминала в открывшемся окне вводится команда

pwd

Эмулятор терминала выведет что-то типа

/home/valerii

2. Вводится команда создания каталога

mkdir 123

т.к. программа выполнена успешно, никаких сообщений выведено не будет. Если произошла ошибка, будет выведено соответствующее ситуации сообщение об ошибке.

3. Проверка создания каталога

ls

Вывод программы ls:

123 Desktop Documents Downloads Public

4. Переход в созданный каталог

cd 123

Не должно быть выведено никаких сообщений.

5. Проверка установки текущим правильного каталога

pwd

Вывод

/home/valerii/123

6. Создание файла

touch z.txt

Сообщений быть не должно.

7. Отображение имени созданного файла

ls

Вывод команды

z.txt

8. Запись в файл z.txt слова Hello

echo Hello > z.txt

Вывода после выполнения команды быть не должно.

9. Отобразим содержимое файла z.txt

cat z.txt

Вывод

Hello

10. Дописать вторую строку с символами C!

echo "C!" >> z.txt

11. Отобразить содержимое файла z.txt

cat z.txt

Вывод

Hello
C!

Задание выполнено.

Удалим созданные файл и каталог.
1. Удалим файл

rm z.txt

Будет выведено предупреждение

rm: удалить обычный файл 'z.txt'?

Необходимо подтвердить удаление нажатием клавиши «y».
2. Выйти из каталога 123 на уровень выше.

cd ..

Вывода не будет.
3. Удалить пустой каталог 123

rmdir 123

Вывода не будет, т.к. каталог пуст.
4. Подтвердить правильность удаления каталога

ls

Вывод

Desktop Documents Downloads Public

Один из парадоксов для начинающего пользователя эмулятора терминала заключается в том, что простые

команды выполнять в эмуляторе терминала дольше, чем в графическом интерфейсе. Однако с повышением уровня владения этой технологией такие операции станет выполнять также просто.

Далее приведен пример, как можно выполнить поставленную ранее задачу в одну команду:

mkdir 123 && touch 123/z.txt && echo Hello > 123/z.txt && echo "C!" >> 123/z.txt

В этом решении использована операция **&&**, которая соответствует **логическому И**. При этом действует правило: если первое (левое) выражение ложно, второе (правое) никогда не будет выполнено.

Проверить содержимое полученного файла можно командой:

cat 123/z.txt

Вывод

Hello
C!

Удалить созданные объекты можно командой:

rm -rf 123

Решение задачи можно записать еще короче.

mkdir 123 && touch 123/z.txt && printf 'Hello\nC!\n' >> 123/z.txt

Существуют еще более лаконичные способы решения

этой задачи, но на начальных этапах освоения приемов работы в эмуляторе терминала лучшим является понятное решение, а не короткое.

В эмуляторе терминала доступны развитые средства обработки текстовых данных. Например, программа **grep** позволяет отфильтровывать строки в соответствии с переданным запросом. Программа **sed** умеет выполнять фильтрацию текста и выполнять изменения в нем. При этом её синтаксис настолько объемен, что его изучение может стать отдельной задачей. Программа **join** позволяет объединять строки из нескольких файлов, являясь по-сути прообразом запроса к табличной базе данных. Программа **printf** выполняет формирование и печать (отображение) данных. Программ **awk** так и называется «язык обработки и поиска шаблонов».

Несмотря на обширные возможности, применение этих программ очень просто на практике и может быть использовано в программах. Например, для замены всех вхождений слова ‘пример’ на ‘привет’ в файле input.txt и сохранение результата в файле output.txt, необходимо выполнить команду

sed ‘s/пример/привет/” input.txt > output.txt

Рассмотрим ряд задач, имеющих практическое значение. Наиболее часто для архивирования и создания резервных копий используются форматы tar, zip, 7z. Они упоминались в пункте 1.4. Приведем пример создания сжатого файла из содержимого текущего каталога с помощью программы **tar**.

tar -cvvf archive.tar ./

Извлечь содержимое из файла archive.tar можно с

помощью команды

tar -xvvf foo.tar

Для формата zip используется следующая команда для создания архива

zip -r archive.zip ./

для извлечения содержимого

unzip archive.zip

Для формата 7-Zip приведен пример архивирования текущего каталога с заданием пароля для шифрования информации.

7z a -pПАРОЛЬ archive.7z

Для извлечения сжатой и зашифрованной информации используется команда

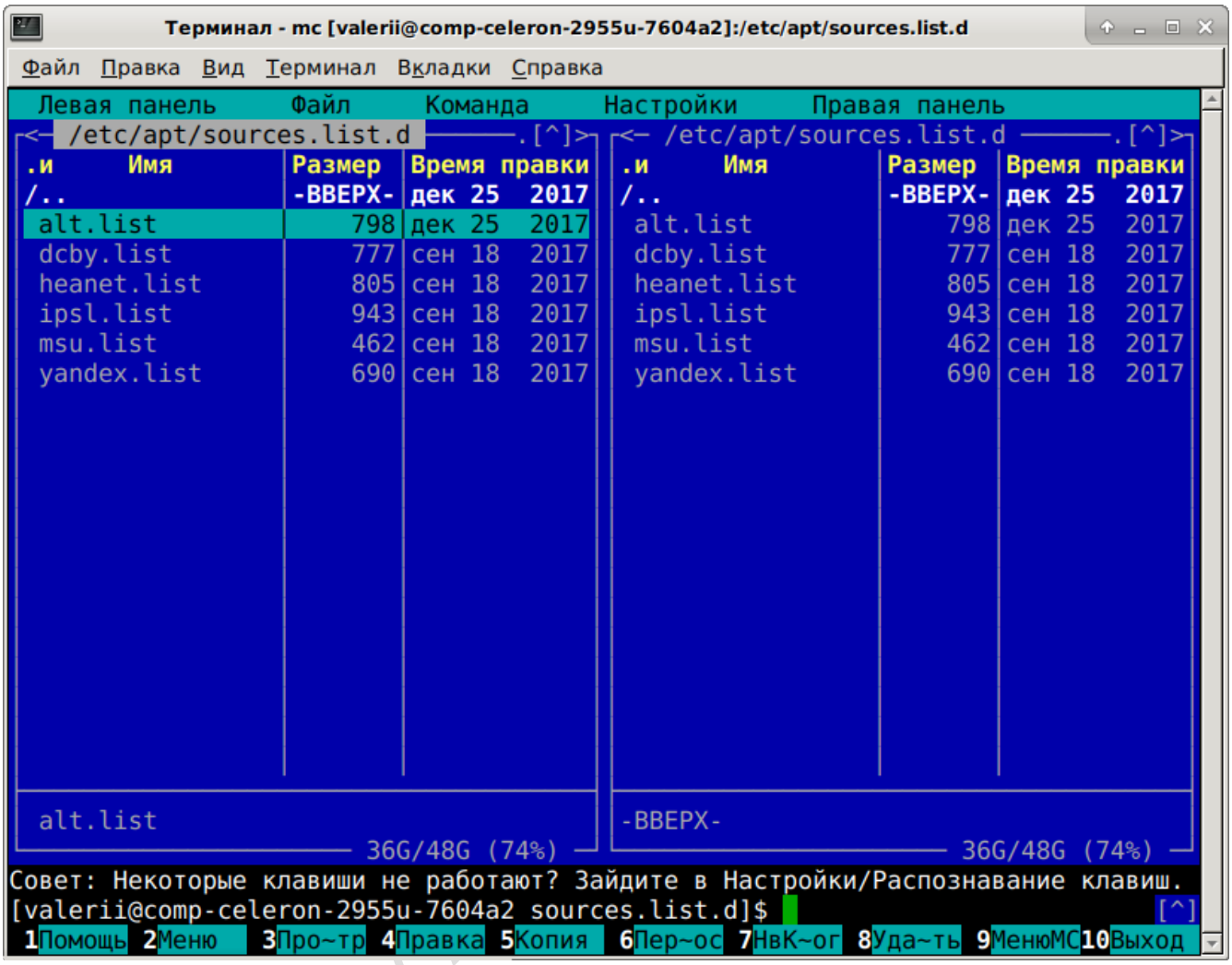
7z x archive.7z

Приведем пример использования ЭТ совместно с визуальной оболочкой **Midnight Commander** для изменения путей к репозиториям в системе apt, о которой упоминалось в пункте 2.1.

Выполните в ЭТ команду

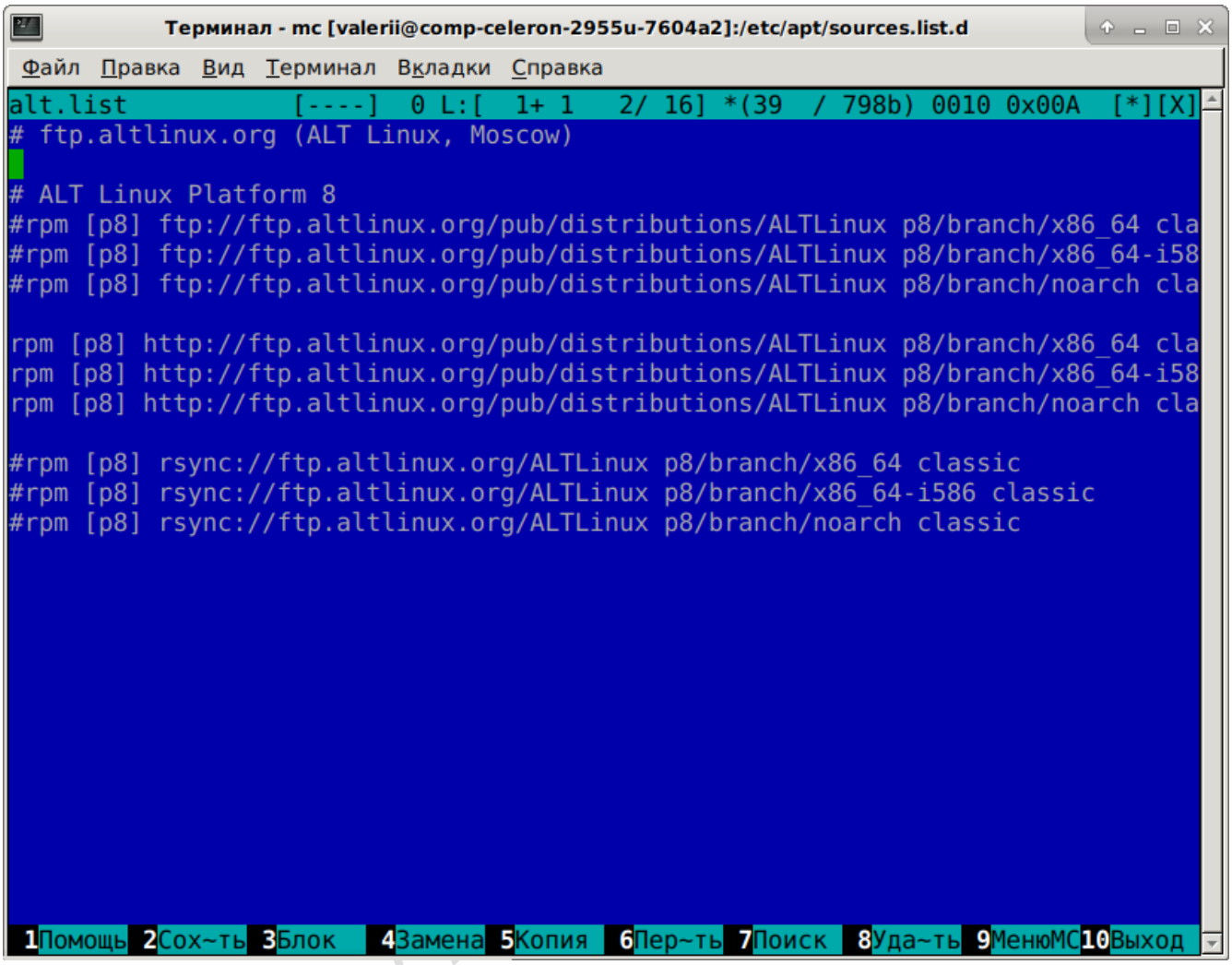
mc

Запуск программы приведет к отображению двух панелей с содержимым текущих каталогов.



Программа имеет интуитивно понятный интерфейс и поддержку манипулятора «мышь». Для перемещения по каталогам используется «мышь» или клавиши клавиатуры стрелка вверх, стрелка вниз, Tab, Enter.

После перехода в каталог **/etc/apt/sources.list.d** необходимо выбрать файл **alt.list** и нажать на клавиатуре клавишу **F4** или щелкнуть левой кнопкой мыши по надписи «**Правка**».



Работа в окне запущенного текстового редактора мало чем отличается от программы Блокнот в ОС WindowsTM. Для добавления дополнительного репозитория необходимо удалить символ «**#**» в начале строки. Для комментирования, наоборот, нужно добавить этот символ в начало строки. После внесения изменений в файл, для их сохранения необходимо нажать клавишу **F2** или щелкнуть мышью по надписи “**Сох~ть**” внизу экрана. Для выхода из текстового редактора используется **F10** или щелчок левой кнопки мыши по надписи «**Выход**».

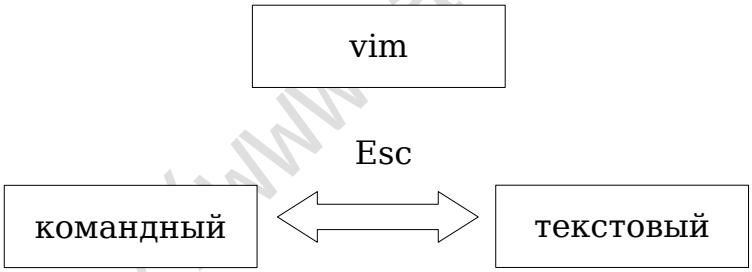
2.4. Текстовый редактор VIM

Для создания текстовых файлов с исходным текстом программ необходим текстовый редактор. В дистрибутиве АльтЛинукс возможно использование различных текстовых редакторов и интегрированных сред программирования. Среди интегрированных сред программирования стоит выделить **KDevelop** и **QTCreator**. Менее требовательной к ресурсам является среда разработки **Geany**. В случае невозможности использования средств разработки необходимо воспользоваться обычным текстовым редактором. При использовании платформ (**QT**, **Gnome**, **XFCE**) с графическим интерфейсом доступны **medit**, **leafpad**. В Midnight Commander встроен собственный текстовый редактор, работа с которым была продемонстрирована в пункте 2.3.

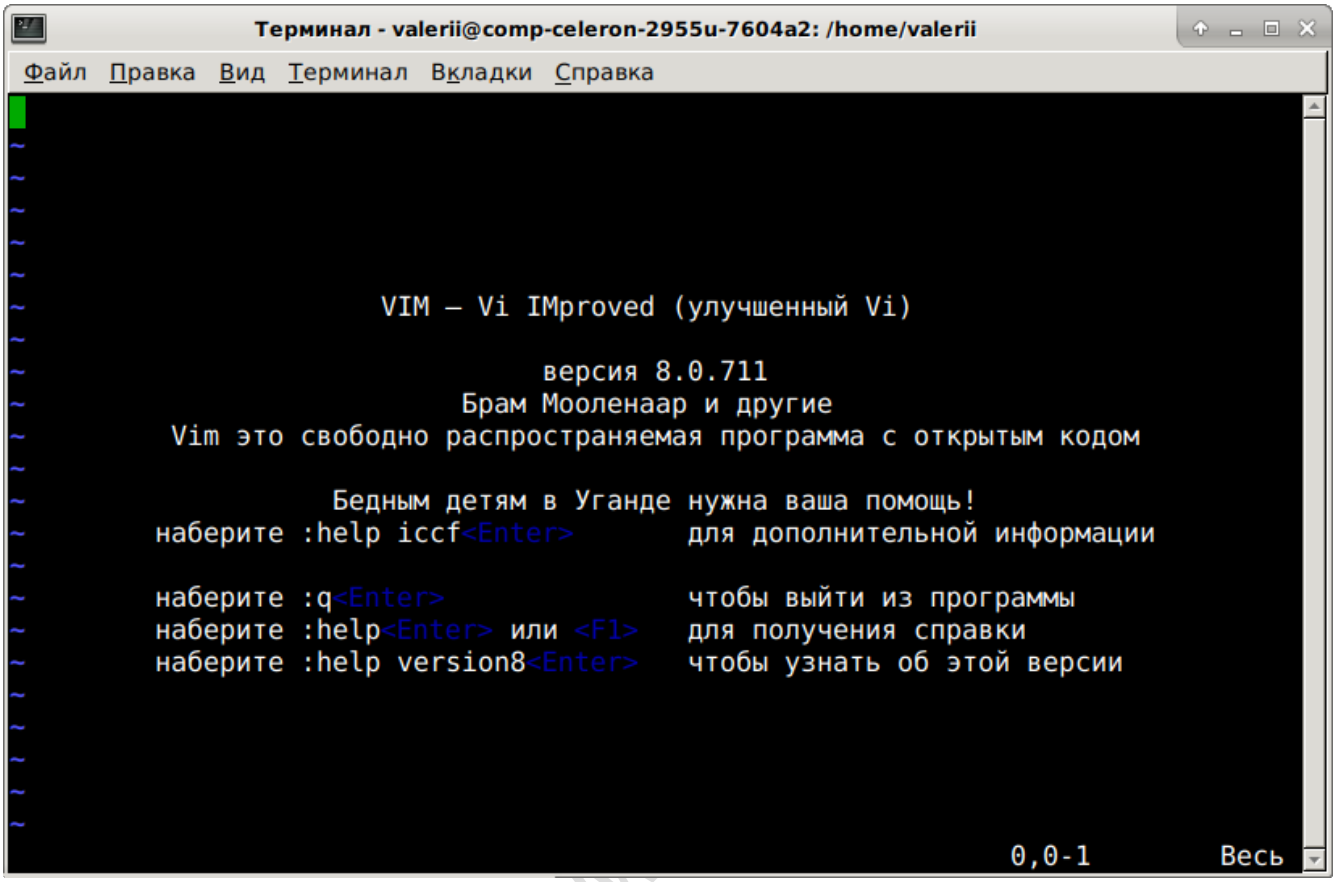
Особенно следует выделить два текстовых редактора **emacs** и **vim**. Emasc является продуктом основателя движения **GNU** и уже давно приобрел статус культового продукта. Это очень сложная в освоении программа, требующая много времени на настройку и установку плагинов. Vim является наследником программы **vi**, которая поставляется со всеми версиями POSIX систем, начиная с 1976 года. Несмотря на прошедшее время и высокого уровня интеграции ПО, при разработке на C постоянно случаются ситуации, самым простым выходом из которых является использование редактора vi или vim.

Для пользователя, имеющего опыт работы только с текстовыми редакторами в графических средах, первое время работать с vi или vim сложно. Это связано с тем, что такая работа требует запоминания текущего состояния программы в каждый момент времени. В графическом режиме для этого используются

подсвеченные кнопки, подсказки, выделение цветом и запоминать состояние программы не требуется. Программы имеют два режима ввода: командный и текстовый. В командном режиме нажатие клавиш на клавиатуре приводит к изменению состояния программы и исполнению команд, соответствующих нажатым клавишам. В текстовом режиме производится вставка символов. Редакторы имеют большое количество «горячих клавиш», которые выполняют действия, редко доступные в графических текстовых редакторах: удаление строки, переход на начало или конец абзаца и др. Для переключения между режимами используется клавиша **Esc**. Со времен использования vim на вычислительных машинах, оснащенных системным динамиком, существует шутка о том, что vim работает в двух режимах: в одном пищит, в другом портит текст.



Редакторы vi и vim имеют одинаковые комбинации «горячих клавиш» и логику работы, но vim более функционален и имеет ряд нужных возможностей, отсутствующих в vi. Целесообразно сначала проверять наличие в системе именно vim и только в случае его отсутствия использовать vi. Далее будет рассматриваться редактор vim без дополнительных плагинов. После запуска командой vim выводится сообщение и текстовый редактор переводится в режим ввода команд.



Для завершения работы с программой без сохранения сделанных изменений необходимо выполнить следующие действия:

1. ввести символ «**:**»
2. ввести символы «q!»
3. нажать Enter

После этих действий работа с редактором будет завершена.

Программа умеет редактировать также и двоичные файлы, поэтому выполнение основных команд демонстрируется на двоичном файле. Редактирование двоичных файлов необходимо для модификации исполняемых файлов программ в двоичном виде, отображаемом с помощью шестнадцатиричных значений и печатаемых символов.

Запуск программы с открытием двоичного файла графического изображения в формате JPEG

vim -b 0010.jpeg

После запуска программа переведена в режим ввода текста. Необходимо нажать клавишу **a**. Удалив несколько символов, необходимо закончить работу без сохранения внесенных изменений с помощью действий, описанных выше.

В командном режиме необходимо отслеживать включенную раскладку клавиатуры, если используются символы, отличные от английских, команды выполнены не будут.

В командах vim часто используются специальные символы, поэтому последовательности команд будут заключены в скобки, т. е. нажатие клавиши **q** будет обозначаться как **(q)**.

Программа vim использует терминологию, несколько отличную от общепринятой. Приведем главные понятия:

- **буфер** – временное хранилище текста, копия отображаемого или редактируемого файла. Каждый редактируемый файл связан с единственным буфером, но каждый буфер может отображаться в неограниченном количестве окон;
- **окно** – область экрана для просмотра и отображения одного буфера;
- **вкладки (сленг. табы)** – механизм группировки и переключения между группами окон;
- **регистр по умолчанию** – наименованное хранилище текста для множественных вставок, наподобие «буфера обмена» в офисных пакетах;
- **именованный регистр** – именованное хранилище текста для множественных вставок. Также напоминает

один из режимов работы «буфера обмена» в офисных пакетах;

- **аббревиатуры** – сокращения, которые при наборе отдельных слов путем замены разворачиваются в соответствующий текст;

- **изменение** – включает в себя вставку текста из регистра, аналог команды «вставка» в графическом текстовом редакторе.

Для получения исчерпывающей справочной информации необходимо после запуска программы ввести команду (**:help**). Для получения справки по конкретной команде (**:help W**).

При открытии файлов с исходными текстами программы возможно изменение режима отображения текста редактором, путем добавления номера строки в файле. Для этого используются команды:

- (**:set number**) — отображать номер строки;

- (**:set nonumber**) — перестать отображать номер строки.

Для того чтобы номер строки отображался всегда, необходимо в домашнем каталоге пользователя в файле ~/.vimrc добавить строку

set number

В случае отсутствия файла ~/.vimrc его необходимо создать.

При открытии файла с текстом возможно как открытие на определенной строке с помощью команды

vim +5 filename

где 5 — номер строки, на которую необходимо переместиться, filename — имя открываемого файла.

При открытии файла присутствует возможность перехода на искомую строку с использованием текстового шаблона

vim +/main filename

где main — название искомой функции, filename — имя файла.

Редактор vim имеет множество **команд перемещения**, часть из которых не встречается в оконных редакторах. Для перемещения используется командный режим. Перемещение на один символ производится расположенными по порядку клавишами ←**h** ↓**j** ↑**k** →**l**. Кнопки — стрелочки позволяют перемещаться в окне. Присутствует поддержка манипулятора «мышь».

Кроме перемещения на один символ возможны следующие варианты перемещения:

- на конец текущего слова вправо (**w**) или влево (**b**);
- на следующее слово вправо (**W**) или влево (**B**);
- в конец следующего слова (**E**);
- на начало (**^**) или конец строки (**$**);
- на начало текущего предложения (**)**) или его конец (**(**);
- на начало текущего абзаца (**{**) или его конец (**}**);
- на первую строку на экране (**H**) или на последнюю (**L**);
- переместится в конец текущего отображения буфера (**G**);
- на первое вхождение произвольного символа (**fv** – переход на первый символ **v** в строке);
- повторить последнюю команду поиска в пределах текущей строки (**;**).

Почти все команды перемещения могут быть повторены произвольное количество раз. Например, **5j** переведет курсор на 5 строк вверх, а **3}** — на три абзаца вниз. Этот способ повторения выполнения команды доступен для большинства команд других типов.

Рассмотрение **команд изменения текста** начнется с команды (**u**) – отмена последнего действия. Набор команд редактирования наиболее удобен для написания исходного кода программы или для редактирования конфигурационного файла и также содержит команды, не встречающиеся в графических текстовых редакторах. Далее приведены команды редактирования текста:

- если vim запущен в ЭТ, вставка из буфера обмена ОС (**ctrl+chift+v**);

- редактирование одного текущего символа (**r**);

- вставить символы с позиции курсора, заменив текущий символ под курсором (**i**);

- добавить текст, начиная с позиции курсора (**a**);

- заменить текст с начала строки (**I**);

- вставить строку до текущей и приступить к её редактированию (**O**);

- вставить строку после текущей и приступить к её редактированию (**o**);

- удалить текущий символ (**d**);

- удалить текущее слово (**daw**);

- удалить текущую строку (**dd**). Команда, удаляющая 10 строк **10dd**;

- удалить символы, начиная от курсора до конца строки (**D**);

- (*) - выделить текущее слово;

- (**/y клавиша Enter**) – подсветит все вхождения символа **y** в текущем окне. Перемещаться между выделенными символами можно с помощью клавиш **N** и **n**;

- (**y**) – копирование в регистр по умолчанию выделенного текста;

- (**yy**) или (**Y**) - копирование всей строки в регистр по умолчанию;

- (**v**) – режим «визуального выделения». До нажатия

клавиши необходимо установить курсор в начало копирования, затем нажать клавишу **v**, далее, перемещаясь по тексту, выделить необходимый фрагмент текста и нажать клавишу (**y**);

- (**/шаблон**) — поиск последовательности символов шаблон;

- (**p**) – вставка текста из регистра по умолчанию;

- (**%s/шаблон/замена**) – заменяет все вхождения слова шаблон на слово замена.

Несмотря на то, что программа vim не имеет графического интерфейса, она способна отображать одновременно буферы нескольких файлов. Команда **vim -o filename1 filename2** откроет два файла, разделенных по горизонтали. Команда **vim -O filename1 filename2** откроет эти же файлы, разделенные по вертикали.

- (**:tabs**) – просмотр всех открытых буферов;

- (**:n**) – переход на предыдущую вкладку;

- (**:N**) – переход на следующую вкладку;

Кроме буферов, vim оперирует понятием вкладки. Команда **vim -p10** откроет 10 пустых вкладок. Максимальное количество открываемых вкладок по умолчанию 10, однако может настраиваться с помощью файла конфигурации. Открытие нескольких файлов во вкладки выполняется командой **vim -p filename0 filename1 filename2**. Для переключения между вкладками используются команды:

- (**:tabs**) – просмотр всех открытых вкладок;

- (**:tabn**) – переход на предыдущую вкладку;

- (**:tabp**) – переход на следующую вкладку;

- (**:tabfirst**) – переход на первую вкладку;

- (**:tablast**) – переход на конечную вкладку;

- (**:tabm 2**) – переход на третью вкладку, потому что нумерация вкладок начинается с 0.

Программа vim способна выполнять сравнение

файлов и выделять отличия цветом. При этом возможно одновременное сравнение двух и более файлов. Например, для сравнения трех файлов необходимо выполнить команду **vim -d filename1 filename2 filename3**. После этого они будут открыты с разделением по горизонтали и подсветкой отличий друг от друга.

Запуск vim с ключом -x, например так, **vim -x filename** приведет к запросу пароля и его подтверждения, после чего при сохранении будет выполнено **шифрование файла**.

Для ежедневного использования возможно использование графического режима редактора vim (**vim -g**). Этот режим требует указания специального ключа на этапе компиляции и сборки программы vim.

Поддержка плагинов позволяет vim полноценно заменять среду разработки программного обеспечения. Перечислим некоторые плагины, облегчающие разработку исходных кодов программ:

- **vundle** – устанавливает плагины с github.com ;

- **neocomplcache** – автоподсказки и автозавершение кода;

- **nerd tree** – навигация по файлам и каталогам, позволяет открыть просмотрщик файлов командой **NERDTree**;

- **nerd commenter** – упрощает добавления комментариев в исходный код программы;

- **ctrlp** – выполняет поиск файла по названию. Для исполнения необходимо нажать комбинацию клавиш ctrl+p и набрать имя файла;

- **vim airline** – повышение наглядности с помощью строки статуса программы vim;

- **taglist** – позволяет просматривать структуру программных файлов, список функций и прочую информацию;

- **supertab** – автодополнение слов нажатием клавиши tab;

- **project** – позволяет организовывать файлы в проекты.

Использование редактора Vim в качестве основного инструмента поднимает престиж программиста среди разработчиков, дает гибкий инструмент для создания программ с минимальным потреблением ресурсов.

Редактор vim можно использовать для форматирования текстовых файлов как стороннюю программу.

Пример изменения кодировки в текстовом файле.

Открыть файл filename в программе vim.

vim filename

Изменить кодировку для отображения имеющегося содержимого файла.

:e ++enc=cp1251

Изменить кодировку текста открытого файла.

:set fileencoding=utf-8

Задать формат последовательности перехода на новую строку (доступные значения dos, unix, mac).

:set fileformat=unix

Сохранение файла в новой кодировке.

:wq

Текстовый редактор vim является полнофункциональным редактором с большим набором функций. Команды vim имеют как полные, так и сокращенные формы записи. Обычно изучение возможностей vim происходит в процессе работы с ним.

2.5. Диагностические утилиты

Команды, доступные в ЭТ, позволяют в интерактивном режиме получать информацию об аппаратном обеспечении вычислительной машины, а также о версиях и настройках ОС и программ. Если после ввода команды получен ответ об отсутствии такой программы, необходимо выполнить её установку Для программы blkid команда установки имеет вид

apt-get install blkid

Для получения информации о микропроцессорах, установленных в системе, используется команда ЭТ

cat /proc/cpuinfo

Команда выводит 26 параметров каждого микропроцессора. Для просмотра определенного параметра необходимо скомбинировать представленную команду с командой grep. Для получения информации о модели микропроцессора можно использовать команду

cat /proc/cpuinfo | grep ‘model name’

Выполнение указанной команды на вычислительной машине, на которой производился набор текста данного пункта книги, привело к следующему выводу:

model name : Intel(R) Celeron(R) 2955U @ 1.40GHz
model name : Intel(R) Celeron(R) 2955U @ 1.40GHz

Таким образом, вычислительная машина оснащена двумя физическими ядрами модели Intel(R) Celeron(R)

2955U @ 1.40GHz. В приведенном выше примере, поток вывода команды cat, которая выводит содержимое псевдофайла cpuinfo файловой системы proc, с помощью символа | передается в поток ввода команды grep, которая выводит строки, содержащие строку ‘model name’, а остальные отбрасывает.

Для получения информации об известных серьезных уязвимостях аппаратного уровня микропроцессора необходимо выполнить команду

cat /proc/cpuinfo | grep 'bugs'

bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass
bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass

Приведенный вывод команды показывает на то, что микропроцессоры обладают серьезными уязвимостями на аппаратном уровне и негодны для хранения информации, составляющей государственную тайну или другие критические данные, но вполне подходят для набора текста, участия в сообществах разработчиков и множества других задач.

Для получения информации о материнской плате используется команда

dmidecode --type baseboard

Для выполнения этой команды необходимо обладать привилегиями суперпользователя, т.е. выполнить команду **su –** и ввести пароль администратора root. Также возможно использование команды **sudo**, однако возможность её использования зависит от настройки

конкретного текущего пользователя. После выполнения команды получения информации о материнской плате возможно ознакомление с информацией о производителе, модели, версии и её серийным номером. Также доступна информация о наличии интегрированных плат дополнения, таких как сетевая плата или видеокарта. Комбинируя команду с утилитой grep, можно получить информацию по одному требуемому параметру.

Команда

dmidecode --type baseboard | grep Manufacturer

отображает информацию о производителе материнской платы

Manufacturer: Packard Bell

а команда

dmidecode --type baseboard | grep -e 'Product Name' -e 'Serial Number'

выводит информацию только о модели материнской платы и её серийный номер

Product Name: EG50_HW
Serial Number: NBNAZ1204003171303311

В приведенном примере команда grep принимает сразу два шаблона для фильтрации и выводит на экран строки как с символами «Product Name», так и с символами «Serial Number».

Для получения информации об установленных планках ОЗУ в слоты материнской платы необходимо

выполнить команду

dmidecode --type 17 | more

Поток вывода команды dmidecode передается на поток ввода команды more, которая выводит информацию высотой в один экран ЭТ и ждет нажатия клавиши пробел для продолжения вывода или q для завершения вывода.

Следует обратить внимание! Команда dmidecode не всегда выводит корректную информацию об аппаратном обеспечении ОЗУ.

Для определения типа микросхем ОЗУ используется команда

dmidecode --type 17 | grep "Type:"

вывод

Type: DDR3
Type: DDR3

Для определения количества свободных слотов для планок памяти ОЗУ используется команда

dmidecode --type 17 | grep "No Module Installed" | wc -l

В приведенной команде в дополнение к перенаправлению потока вывода dmidecode в поток ввода grep, поток вывода grep перенаправляется в поток ввода wc. Программа wc может выполнять подсчет количества строк, слов и символов в файлах и данных, полученных через поток ввода.

Не весь объем памяти ОЗУ может быть доступен по

причине выхода из строя аппаратного обеспечения или неверной настройки ОС. Информация об объеме фактически доступной ОЗУ получается с помощью команды

cat /proc/meminfo | grep -e MemTotal -e MemFree

её вывод

MemTotal: 10109896 kB
MemFree: 8194740 kB

Особенность приведенной команды в отсутствии кавычек в параметрах команды grep. Они отсутствуют, так как передаваемые шаблоны для поиска не имеют пробелов или других разделительных символов.

Информация о **распределении прерываний** не считается критически важной для пользователя вычислительной машины, но в некоторых случаях неправильное распределение прерываний приводит в некоторых режимах к замедлению работы вычислительной машины. Для получения информации о прерываниях используется команда.

cat /proc/interrupts

Немного подробнее о концепции прерываний написано в пункте 1.3.

Информация об установленных платах расширений доступна с помощью команды **lspci**. Она выводит информацию об установленных видеокартах, сетевых платах, USB, ISA, SATA контроллерах и прочее. Например, для получения информации об интегрированной видеокарте необходимо выполнить

команду

lspci | grep VGA

пример вывода которой приведен ниже

00:02.0 VGA compatible controller: Intel Corporation Haswell-ULT Integrated Graphics Controller (rev 0b)

Информация об устройствах, использующих интерфейс USB, доступна с помощью команды

lsusb

которая выводит общую информацию об устройствах, например

Bus 001 Device 002: ID 8087:8000 Intel Corp.

Добавление ключа к команде следующим образом **lsusb -t** выводит информацию об устройствах в виде древовидного списка.

Для получения подробной информации используется команда

usb-devices

Далее рассмотрено получение информации об установленных накопителях информации, в том числе жестких дисках. Узнать, какие накопители подключены в ОС, можно с помощью команды

lsblk

она предоставляет информацию в наглядном виде с помощью символов псевдографики

```
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0   223,6G  0 disk
├─sda1   8:1    0   49,2G   0 part /
└─sda2   8:2    0   174,4G  0 part /home
sdb      8:16   0   119,2G  0 disk
└─sdb1   8:17   0   119,1G  0 part
```

В соответствии с выводом программы lsblk вычислительная машина оснащена двумя накопителями, способными сохранять 223,6 Гб и 119,2 Гб. Также указаны точки монтирования. Для раздела sda1 точкой монтирования является корневой каталог /, а для раздела sda2 точка монтирования /home . При каждой установке ОС или в процессе её эксплуатации ОС точки монтирования и набор разделов может меняться. Более подробную информацию о разделах выведет команда **blkid** .

Информацию о доступных разделах на носителях информации также можно получить с помощью команды

fdisk -l

в конце вывода команды будут примерно такие строки

Устр-во начало Конец Секторы Размер Тип
/dev/sda1 2048 103151615 103149568 49,2G Файл.система Linux

Для получения информации по технологии S.M.A.R.T.

используется программа **smartctl** . Для получения краткой справки по работе с программой используется ключ **-h** .

Узнать модель накопителя можно с помощью команды

smartctl -i /dev/sda

Указанная команда отобразит модель устройства, серийный номер, версию прошивки устройства и прочую информацию.

Для получения всей информации об устройстве используется команда

smartctl --all /dev/sda

В случае отсутствия поддержки технологии S.M.A.R.T. выведенные значения будут не корректными.

Важной задачей с точки зрения обслуживания вычислительной машины является проверка температуры основных компонентов. Для определения температуры, сообщаемой сенсором центрального процессора, используется команда

acpi -t

Также можно использовать команду

sensors

пример вывода

coretemp-isa-0000
Adapter: ISA adapter

Package id 0: +46.0°C (high = +100.0°C, crit = +100.0°C)
Core 0: +42.0°C (high = +100.0°C, crit = +100.0°C)
Core 1: +40.0°C (high = +100.0°C, crit = +100.0°C)

Для определения температуры накопителей информации используется команда **hddtemp**, применить её можно, например, так **hddtemp /dev/sda** . Программа выдаст шуточный комментарий и отобразит полученное значение

/dev/sda: SSD Smartbuy 240GB #�: 30°C или °F

или так

/dev/sdb: OCZ-AGILITY4 #�: нет датчика

В процессе загрузки и работы вычислительного устройства ОС ведет протоколирование основных действий с аппаратным обеспечением на уровне ядра. Доступ к этим записям можно получить с помощью команды **dmesg**. Например, чтобы получить записи о сетевом интерфейсе eth0, необходимо выполнить команду

dmesg | grep eth0

которая выведет следующие сообщения

[4.324760] tg3 0000:01:00.0 eth0: Tigon3

[partno(BCM57786) rev 57766001] (PCI Express) MAC address 20:1a:06:76:04:a2

[4.324764] tg3 0000:01:00.0 eth0: attached PHY is 57765 (10/100/1000Base-T Ethernet) (WireSpeed[1], EEE[1])

[4.324766] tg3 0000:01:00.0 eth0: RXcsums[1] LinkChgREG[0] MIirq[0] ASF[0] TSOcap[1]

[4.324768] tg3 0000:01:00.0 eth0: dma_rwctrl[00000001] dma_mask[64-bit]

[5.349252] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready

[5.520883] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready

Для целей аппаратной автоматизации или подключения датчиков одним из популярных интерфейсов остается последовательная шина (**Serial**) и интерфейс **UART**. Для работы с устройствами, подключаемыми с помощью неё, основными инструментами являются **setserial** , **stty** и **minicom**. После установки вызов программы minicom выполняется следующим образом **minicom -s** . Более подробную информацию необходимо искать в литературе, посвященной этим интерфейсам.

Для определения доступных разрешений экрана используется команда **xrandr** . Кроме разрешений экрана она выводит состояние интерфейсов, поддерживаемых микропроцессором видеокарты. Следует учесть, что не все выводы могут присутствовать физически, т.к. производители могут сокращать их количество с целью уменьшения стоимости конечного изделия, отсутствия необходимого физического пространства или по другим причинам.

Кроме вывода конфигурационной информации xrandr

также может поворачивать изображение вправо или влево, например **xrandr -o left** или вернуть его в нормальное положение **xrandr -o normal** .

При наличии клавиатуры с подсветкой, необходимо использовать команды для её включения или выключения.

xset led 3 – включает подсветку клавиатуры.

xset -led 3 – выключает подсветку клавиатуры.

В заключении раздела приведем несколько приемов работы, которые значительно проще реализовать с помощью ЭТ.

С целью сохранения важной информации желательно периодически выполнять затирание всей поверхности носителя информации. Для некоторых типов устройств заполнение нулями не является гарантией от стирания информации. Жесткие диски могут сохранять остатки информации на своих дисках до 7 и более циклов перезаписи, однако для бытовых целей достаточно и однократной перезаписи носителя информации. Используя программу dd, перезапись поверхности носителя информации sda можно осуществить выполнив следующую команду

dd if=/dev/urandom of=/dev/sda

Приведенная команда предписывает утилите dd получать информацию от псевдоустройства /dev/urandom, являющегося программным генератором случайных чисел, и записывать случайные значения в устройство /dev/sda до исчерпания свободного пространства устройства /dev/sda. После перезаписи устройства таким образом, его необходимо заново подготовить к работе с ОС, выполнив разбивку на логические разделы. Для этого используются программы fdisk, cfdisk или программа с

графическим интерфейсом **gparted**.

Пользователи, начинающие знакомиться с ОС Линукс, часто параллельно используют несколько ОС. Если для выбора загрузки ОС используется загрузчик grub и по какой-то причине загрузка в ОС, отличная от Линукс, не производится, достаточно выполнить команду **update-grub** в ЭТ с привилегиями суперпользователя. Команда обновит список доступных для загрузки ОС и их конфигурации.

Для настройки аппаратного счетчика времени используется программа **hwclock**. Время, используемое в ОС, можно посмотреть и изменить с помощью программы **date**. Для отображения календаря на текущий месяц используется команда **cal**, а чтобы посмотреть календарь за 2000 год, необходимо использовать команду **cal -y 2000**.

Для задания временной зоны и аппаратного времени необходимо обладать правами root.

Примеры приемов работы с hwclock:

- **hwclock –set ln /…./Moscow /etc/localtime** – задание временной зоны;
- **hwclock -r** узнать аппаратное время;
- **hwclock --systohc –utc** — синхронизация аппаратного времени с utc;
- **hwclock --systohc** — синхронизировать аппаратное время с системным;
- **hwclock --hctosys** — синхронизировать системное время с аппаратным.

Хранение информации на оптических дисках сегодня используется достаточно редко. Однако при возникновении такой потребности все операции можно провести с помощью программ **genisoimage** и **wodim**.

Для просмотра списка устройств выполняющих запись оптических дисков и информации об их

совместимости со стандартами MMC, используется команда

wodim -prcap

Просмотр списка устройств на шинах SCSI и IDE, их модели и точек монтирования используется команда

wodim --devices

Для открытия или закрытия лотка оптического диска используются команды:
- открытие лотка

eject

- закрытие лотка

eject -t

Полезная информация о настройке ОС доступна при вызове команды

ulimit -a

2.6. Диагностические сетевые утилиты. Сетевые службы

По сравнению с 1976 годом, когда появилась первая реализация стека TCP/IP, простота настройки сетевых соединений значительно повысилась. Обычно конфигурирование установленных сетевых карт производится в процессе установки ОС. Для получения первоначальной информации о конфигурации сетевых устройств и подключений используется конфигуратор сетевых соединений программа ifconfig.

Выполнение команды **ifconfig -a** в ЭТ приведет к отображению основной информации об имеющихся интерфейсах. Для перечисления доступных интерфейсов используется команда **ifconfig -a | grep Link** . Вывод этой команды представлен ниже:

eth0 Link encap:Ethernet HWaddr 24:8C:80:26:19:C0
lo Link encap:Local Loopback
wlan0 Link encap:Ethernet HWaddr 12:20:A3:F3:A1:09

В соответствии с выводом программы, вычислительная машина имеет три сетевых интерфейса. **eth0** – интерфейс, связанный с сетевой картой, передающей данные по **8P8C**, т.е. по витой паре. При наличии двух таких сетевых карт, второй интерфейс имел бы обозначение **eth1** и т.д. Следующий интерфейс **lo**. Его обозначение является сокращением от loopback, т.е. кольцо. Он имеет связь только с одним сетевым адресом **127.0.0.1** и служит для обмена информацией между процессами внутри одной вычислительной машины, а

также для отладки сетевых приложений на этапе их написания. Данный интерфейс не имеет аппаратной реализации и реализован с помощью центрального микропроцессора и ОЗУ, поэтому скорость его работы сопоставима с работой с каналами и другими приемами межпроцессовых взаимодействий. Обозначение **wlan0** соответствует сетевому соединению по одному из беспроводных интерфейсов (**WIFI**).

Для получения подробной информации о соединениях WIFI, используется программа **iwconfig** .

wlan0 IEEE 802.11 ESSID:off/any
Mode:Managed Access Point: Not-Associated Tx-Power=off
Retry short limit:7 RTS thr:off Fragment thr:off
Encryption key:off
Power Management:off

Пакет для работы с wifi в ЭТ называется **wireless-tools**. В него входят такие программы как **iwlist** для получения информации о соединении и **iwspy** для сбора статистики по конкретному узлу, использующему беспроводную сеть.

Отдельным классом сетевых интерфейсов является **Bluethoot**. Он занимает промежуточное место между аппаратными интерфейсами наподобие стека интерфейсов USB и беспроводными интерфейсами WIFI. Его соединения не отображаются совместно с универсальными сетевыми интерфейсами, т.к. основное его назначение обмен информацией с устройствами, находящимися в непосредственной близости. Конфигуратором bluethoot устройств является программа **hciconfig** , которую необходимо устанавливать дополнительно.

Для взаимодействия между конечной вычислительной машиной и устройством маршрутизации или сервером используются уникальные физические адреса сетевых устройств, **MAC** – адреса (**Hwaddr** – в приведенном выше выводе). Однако на сегодняшний момент часть производителей позволяет произвольно менять MAC – адрес в соответствии с потребностями пользователя. При изменении MAC — адреса необходимо отслеживать его уникальность в пределах подсети до первого маршрутизатора.

Для обмена информацией в разных подсетях используется сетевой адрес или **IP**. Его уникальность может обеспечиваться специальными технологиями, например **NAT**. Технология NAT подменяет IP адреса вычислительных машин, маршрутизацию которых она обеспечивает, на связку из собственного IP адреса и номера порта. Получая данные из сети на определенный номер порта, устройство с NAT, используя свои таблицы соответствий, пересылает данные адресату с внутренним IP адресом.

При подключении к вычислительной сети автоматическое получение IP адреса и других настроек обеспечивает технология **DHCP**. Принцип работы этой технологии заключается в том, что при включении вычислительного устройства в сеть, устройство производит широковещательную рассылку сообщения специального формата. В ответ на это сообщение DHCP – сервер высылает конфигурационную информацию, содержащую сетевой адрес, маску сети.

В случае необходимости получения доступа к подсетям с несколькими масками сети с помощью одного интерфейса используется следующая команда

ifconfig eth0:0 192.168.12.89 up

Команда предписывает к интерфейсу **eth0** добавить псевдоним **eth0:0**, который использует сетевой адрес **192.168.12.89**, соответствующий сетевой маске **255.255.255.0** и подсети **192.168.12.0**. При этом основной сетевой адрес, сетевая маска и подсеть может быть любым другим.

Защиту от нежелательных подключений обеспечивают программы фильтрации сетевых пакетов, часто называемые файерволы. Наиболее популярной программой является **iptable**. Он имеет сложный язык настройки. Неправильно составленные правила iptable могут сильно замедлить работу сетевых интерфейсов или заблокировать работу системных программ, обеспечивающих базовую функциональность. Приведем пример открытия порта 80, который по умолчанию используется для передачи данных по протоколу http. Для этого необходимо выполнить ряд действий.

1. Выполнить команду добавления нового правила фильтрации

iptables -I INPUT -p tcp --dport 80 -m state --state NEW -j ACCEPT

2. Сохранить внесенные изменения

service iptables save

3. Перезагрузить iptables

/etc/init.d/iptables restart

Для просмотра открытых портов используется команда

iptables --line-numbers -n -L

Для получения статистики сетевых подключений используется программа **netstat**. Программа при выполнении отправляет в поток вывода от десятка до сотен строк, поэтому её вывод здесь представлен не будет. Список всех открытых портов TCP можно получить с помощью команды **netstat -at** . Для просмотра открытых портов по протоколу UDP **netstat -au** . Список только прослушиваемых TCP портов **netstat -lt** . Список всех открытых портов **netstat -s** . Для отображения списка открытых портов с указанием процессов, которые их используют **netstat -p** .

Для получение информации о сетевых соединениях также может использоваться еще одна программа **ss**. Для отображения списка процессов с открытыми соединениями с помощью этой программы используется команда **ss -p** . Команда отображения списка сокетов для прослушивания **ss -l** .

Программа отображения списка открытых файлов **lsof** также отображает список сетевых соединений. Для отображения всех сетевых соединений используется команда **lsof -i** , а для отображения списка процессов, работающих с портом 80, используется команда **lsof -i : 80** .

В некоторых случаях возникает необходимость использовать одно подключение для обмена информацией с устройствами, находящимися в разных подсетях и имеющих несовместимую маску подсети.

В больших компьютерных сетях часто используются **прокси-сервера**, работа которых напоминает действия DHCP-серверов. Кроме того, прокси-сервера могут выполнять фильтрацию сетевых соединений, отбрасывая

опасные или не разрешенные пакеты и подключения. Для получения доступа к вычислительным машинам, расположенным за прокси-сервером, в терминале необходимо выполнить команду

export http_proxy='http://192.168.12.1:3128'

Команда создает переменную окружения **http_proxy**, которая содержит сетевой адрес и порт прокси-сервера. Для указания имени пользователя и пароля используется следующая конструкция команды

export http_proxy='http://USERNAME:PASSWORD@PROXY_IP:PROXY_PORT/'

где

USERNAME – имя пользователя среди учетных записей пользователей прокси-сервера.

PASSWORD – пароль пользователя, соответствующий имени пользователя.

PROXY_IP – сетевой адрес прокси-сервера.

PROXY_PORT – порт прокси-сервера, который обслуживает подключения из внутренней вычислительной сети.

Созданная таким образом переменная окружения будет существовать только в ЭТ, в котором выполнена команда. При повторном открытии терминала, необходимо повторно выполнить команду. Команду можно дополнить вызовом ЭТ, поместить в текстовый файл, изменить расширение файла на sh и дополнить его права правом на исполнение. Существуют и другие способы задания переменных окружения по умолчанию.

Для проверки доступности любого другого сетевого

устройства существует утилита для выполнения эхо-запросов **ping**. Эхо-запросы описаны ранее в пункте 1.8. Необходимо учесть, утилиты, подобные ping, существуют для большинства протоколов, однако их название может немного отличаться. Приведем вывод команды **ping 127.0.0.1 -c 2** . Ключ **-c 2** указывает программе о необходимости отправить 2 запроса.

PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_req=1 ttl=64 time=0.045 ms
64 bytes from 127.0.0.1: icmp_req=2 ttl=64 time=0.049 ms

--- 127.0.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1052ms
rtt min/avg/max/mdev = 0.045/0.047/0.049/0.002 ms

После информации о результатах каждого запроса выводится общая статистика о выполненных эхо-запросах.

Программа **ping** отправляет запросы с небольшим интервалом, однако, если добавить ключ **-f** , интервал будет равен 0, что может идентифицироваться сетевыми устройствами как флуд-атака. В результате источник может быть заблокирован или ограничен в привилегиях. Ключ **-f** доступен только для пользователей с привилегиями администратора вычислительной машины.

Для получения информации о промежуточных узлах вычислительной сети используется утилита **traceroute**, которая формирует сообщения ICMP протокола с типом 11 и постоянно увеличивающимся числом переходов TTL. В соответствии с протоколом IP, в случае отбрасывания

пакета из-за обнуления поля TTL, промежуточный узел должен отправить обратный пакет с сообщением об уничтожении пакета. Таким образом, отправив сообщение с TTL=1, источник узнает сетевой адрес следующего за ним в сетевом узле. Сообщение с TTL=2 приводит к получению сетевого адреса второго промежуточного узла и т.д. Следует учитывать, что на некоторых узлах службы протокола ICMP могут быть отключены. Также замечено, сетевой трафик, попадающий в РФ, лишается большей части служебной информации. Это легко проверить, выполнив запросы с территории РФ и такие же запросы с территории стран НАТО.

Для сбора пересылаемой и получаемой информации используется программа **tcpdump**. Работа с ней производится с привелегиями администратора вычислительной машины. Для просмотра интерфейсов, доступных для сбора информации, используется команда **tcpdump -D** . Далее приведен вывод этой команды.

1.eth0
2.any (Pseudo-device that captures on all interfaces)
3.lo

Для сбора всех пересылаемых пакетов через интерфейс eth0 используется команда **tcpdump -i eth0** . Если необходимо собрать максимальное количество информации, используется команда **tcpdump -v -i eth0** . Данные, переданные через сетевые интерфейсы и представленные в текстовом виде, генерируют значительные объемы информации, которые проскакивают по экрану бесконечным потоком. Одно из решений этой проблемы — запись потока в файл для последующего анализа **tcpdump -i eth0 -w file.txt** .

Просмотр собранной информации удобно производить с помощью этой же утилиты, выполнив команду **tcpdump -r file.txt** .

Для непосредственного наблюдения сетевой активности необходимо выполнять предварительную фильтрацию поступающей информации. Далее приведены примеры фильтрации:

- **tcpdump -i eth0 ip dst 127.0.0.1** - просмотр пакетов от одного ip;
- **tcpdump -i eth0 ip src 127.0.0.1** - просмотр отправляемых пакетов;
- **tcpdump -vv -i eth0 port 514** - дампирование порта;
- **tcpdump -vvv -i eth0 udp** - дампирование информации, передаваемой по протоколу UDP;
- **tcpdump -i eth0 net 192.168.1.1** - дампирование пакетов подсети;
- **tcpdump -i eth0 less 50** - дампирование пакетов интерфейса размером менее 50 бит;
- **tcpdump -i eth0 greater 100** – дампирование пакетов больше 100 бит;
- **tcpdump -A -i eth0** - просмотр содержимого пакетов;
- **tcpdump host name** – просмотр отправляемых и получаемых пакетов между источником и узлом name;
- **tcpdump ip host name2 and not name** – просмотр всех пакетов между узлом name2 и всеми другими узлами за исключением узла name;
- **tcpdump udp and not src net localnet** – отслеживать UDP-пакеты, приходящие от узлов с сетевыми адресами, не соответствующими локальной маски сети.

Более гибким и наглядным инструментом сбора информации о сетевой активности является программа

Wireshark или её аналог без графического интерфейса **tshark**.

Приведем несколько команд программы **tshark**:

- **tshark -D** – выводит список доступных интерфесов;

- нажатие на клавиатуре **Ctrl + C** – прекращение выполнения команды;

- **tshark -i 2 -w test.pcap** – получение пакетов сетевого интерфейса 2 и их запись в файл **test.pcap**;

- **tshark -i 3 icmp** – получение пакетов ICMP протокола сетевого интерфейса 3;

- **tshark -r test.pcap -V | more** - просмотр ранее записанного файла test.pcap в постраничном режиме.

Одним из приемов отправки сообщений в сеть является их отправка из ЭТ. Однако эта возможность доступна, начиная с версии 4.4. При этом данная возможность должна быть указана при сборке исполняемых файлов. Текущей версией, поставляемой вместе с дистрибутивом, является 3.2.57. Приведем примеры команд.

Отправка текстового запроса на основе вывода команды echo с использованием протокола tcp, используя псевдоустройства

echo "GET / HTTP/1.0" > /dev/tcp/192.168.3.20/80

Отправка текстового запроса на основе вывода команды printf

printf "GET / HTTP/1.0" > /dev/tcp/192.168.3.20/80

Для отправки двоичных данных необходимо использовать файл с данными, так набор двоичных данных с помощью клавиатуры не предусмотрен.

Отправка 160 байт в двоичном виде по tcp порт 7654

с использованием утилиты **dd** для передачи данных для отправки

dd if=binary.dat bs=160 count=1 > /dev/tcp/192.168.3.20/7654

Другим приемом, не зависящим от версии bash, является использование программы **netcat** (**nc** – сокращенная форма записи вызова программы).

Отправка текстового запроса на основе вывода команды echo с использованием программы netcat

echo -e "GET / HTTP/1.0\n\n" | nc 192.168.3.20 80

Утилита netcat также может выступать в качестве программы, получающей данные из вычислительной сети.

Для получения информации по протоколу UDP на порт 5555 используется команда

nc -lu 5555

Для отправки данных на порт, открытый предыдущей командой, используется команда

echo “Hello” | nc -u 127.0.0.1 5555

Приведенная команда передаст информацию из окна ЭТ с командой отправки информации, в другое окно ЭТ с командой открытия и прослушивания порта.

Программа netcat так же умеет письма по протоколу SMTP и сканировать порты.

Распространенной задачей является определение процесса, занимающего или слушающего определенный

сетевой порт. Для этих целей можно использовать различные приемы, однако чаще применяется программа **netstat**. Знание трех способ вызова программы netstat обычно достаточно для определения причины недоступности порта:

- **netstat -anpu** — отображение загруженных сервисов с открытыми портами;
- **netstat -tlnp** — отображение процессов с открытыми портами;
- **netstat -a** — отображение всех открытых портов без подробностей.

Для определения установленных соединений достаточно выполнить команду **arp**, которая позволяет выполнять манипуляции с кешем протокола arp. Отображение аппаратных адресов выполняется командой.

arp -a mac

Для получения информации о версии ОС ЭВМ, подключающийся к вашему устройству, можно использовать пассивный сканер **p0f**, который требует запуска с правами администратора. После запуска программы, она начинает отслеживать формат принимаемых IP пакетов и пытается таким образом определить версию ОС отправителя. Этот сканер не всегда может определить версию ОС, однако против него бессильны практически все анонимайзеры, т. к. они не изменяют структуру пересылаемого IP пакета.

Перечислим программы, ознакомление с которыми является необходимым минимумом для работы в сети:

- **write** – отправляет сообщение или файл указанному адресату;
- **rsync** – синхронизация локальных и удаленных

каталогов;

- **wget** – скачивание файлов и статических сайтов, независимо от разрывов соединения с возможностью докачивания через некоторое время;

- **openssh** – программа удаленного доступа с помощью ЭТ по зашифрованному каналу;

- **nmap** – получение списка открытых портов;

- **whois** – получение информации о владельце ресурса по протоколу whois;

- **nikto** – сканер уязвимостей серверов, работающих с использование протокола http;

- **snort** – автоматический анализатор / блокировщик сетевых пакетов;

- **awstats** – в режиме реального времени анализ статистики вэб-серверов.

2.7 Информационная безопасность

Настали времена, когда государственные службы безопасности всех стран перестали скрывать свое пристальное внимание к вычислительным сетям. Чем чаще и интенсивнее человек использует глобальные информационные сети, тем выше риск нанесения ему ущерба в том или ином виде, т.е. меньше его безопасность. Таким образом, кажется ответ очевиден: не используя вычислительные сети человек не подвергается опасности. **Это не так!** Данные о человеке есть в вычислительных сетях независимо от того, использует ли он её сознательно или нет, а значит каждый человек подвергается опасности. На текущий момент, в среднем, ущерб, наносимый человеку при использовании вычислительной сети меньше, чем польза от преимуществ, получаемых им от их использования.

Сфера информационной безопасности является профессиональной областью деятельности, связанной с реальным риском для здоровья и жизни. Для минимизации рисков и угроз безопасности при работе в вычислительных сетях необходимо знать ряд концепций и понимать базовые принципы работы в них.

1. Стать высококвалифицированным специалистом в области информационной безопасности, читая книги и сидя за компьютером, невозможно!

2. Чем дольше человек присутствует в вычислительной сети, тем сложнее оставаться анонимным, независимо от уровня квалификации человека. Чем выше квалификация, тем дольше и лучше сохраняется частичная анонимность.

3. Значительную часть угроз несут сознательные действия некоторых участников обмена информацией в компьютерных сетях, т.е. правонарушения и

преступления. Структура преступности в сфере информационных технологий полностью повторяет любую другую преступность. Причины преступности изучает наука криминология. Если социум не подвержен информационному и культурному давлению, количество людей, сознательно преступающих закон, не превышает 3%, даже во время войн и голода. На сегодняшний день давление на социум осуществляется с использованием глобальных вычислительных сетей.

4. Невозможно продолжительное время заниматься сомнительной или противозаконной деятельностью в сети Интернет, не попав в поле зрения служб безопасности.

5. Бытовые устройства, подключенные к вычислительной сети, опасны, т.к. изначально проектируются для сбора информации на самой защищенной территории человека – в его доме!

6. Проблема использования нелицензионного программного обеспечения и информационного контента в масштабах страны напрямую связано исключительно с общим достатком, а не с культурными различиями или политическим строем.

7. Потеря контроля над своими персональными данными является угрозой безопасности для каждого человека. Подразумевается свободная передача персональных данных между государственными структурами, корпорациями, частным бизнесом, конкретными персоналиями.

8. Частный случай предыдущего пункта, лицензионные соглашения о передачи обезличенных данных третьим лицам. Причина опасности использования соглашения о передаче данных третьим лицам в том, что с увеличением объема информации она перестает быть обезличенной и снова становится

персональной. Именно поэтому оставаться анонимным в сети продолжительное время невозможно в принципе.

Основу информационной безопасности составляет ряд правил, базирующихся на общих правилах безопасности, которые в первую очередь опираются на здравый смысл.

1. Стоимость защищаемой информации всегда должна быть меньше стоимости получения доступа к ней.

2. Принцип «Неуловимый Джо». Встречаются два ковбоя. Мимо проходит третий. Первый говорит второму: «Это неуловимый Джо прошел.» Второй спрашивает: «Такой крутой? Поймать никто не может?». Первый отвечает: «Нет. Ненужен никому».

3. Бесплатный сыр бывает только в мышеловке.

4. Не бывает облачных технологий, бывают чужие сервера. (Пословица системного администратора одной компании). Эта пословица относиться как к бесплатным, так и платным почтовым серверам, социальным сетям, файловым хранилищам и всем другим Интернет-сервисам. Облачные сервисы дешевле, потому что они менее безопасны.

5. Если какую-то информацию можно не публиковать в вычислительной сети, её нужно НЕ публиковать.

6. Если какую-либо информацию следует опубликовать, подумайте как к ней отнесется каждый член Вашей семьи сегодня, через 15 лет, через 30 лет. Как к этой публикации отнесутся Ваши дети и другие родственники.

7. Собака лает, караван идет. Если по какой-то причине Вам приписывается информация, к которой Вы не хотите иметь никакого отношения, выводите общение по этим вопросам в правовое поле. Все другие варианты для ценителей Сунь-цзы.

8. Следовать правилам безопасности необходимо

постоянно.

Не следует забывать, что лучшими массовыми бесплатными школами в области безопасности являются ДОСААФ РФ и ВС РФ.

3. Язык программирования Си
3.1. Концепция языка. Первая программа

Язык программирования Си позволяет описывать алгоритмы в текстовом виде, с целью преобразования полученных текстовых файлов, написанных на языке Си, в набор инструкций понятных микропроцессору, в формате исполняемого файла целевой операционной системы. Язык Си стандартизирован. Актуальным на сегодняшний день является стандарт языка, описанный в документе ISO/IEC 9899:2018. Сокращенное название стандарта C18. Мое личное мнение заключается в том, что прочитать стандарт и изредка перечитывать необходимо, но знание стандарта наизусть не является обязательным. Для этого есть ряд причин:

1. Стандарт разрабатывается по частям консорциумом, а программист — это один человек.
2. В моей практике не было встречено ни одного популярного компилятора, который полностью реализовал стандарт языка.
3. Срок жизни конкретной версии стандарта языка не более 10 лет, т. е. стандарт однозначно придется переучивать.
4. Компиляторы постепенно реализуют возможности стандарта от версии к версии. Таким образом, текст, написанный для одной версии компилятора, может некорректно обрабатываться компилятором другой версии, причем таких расхождений со стандартом могут быть десятки. Со временем они накапливаются в тексте и текст быстрее и проще написать заново, чем разобраться во всех доработках.
5. Некоторые разработчики компиляторов умышленно включают элементы, не соответствующие стандарту языка, делая исходный текст программ

зависимым от своей версии компилятора.

Язык Си — это инструмент, позволяющий ускорить написание программ и повысить их переносимость между аппаратными платформами, по сравнению с ассемблером. Такой эффект достигается за счет абстрагирования от задач, связанных с непосредственным управлением элементами и блоками микропроцессора. Си относится к высокоуровневым языкам программирования, но остается достаточно низкоуровневым, чтобы с его помощью реализовывать высокоэффективные программы.

В среде специалистов, не использующих Си как основной инструмент разработки, распространяется мнение о сложности языка и высокой цене ошибки. На практике язык Си можно использовать как алгоритмический язык, так и инструмент разработки драйверов и других программных компонент, взаимодействующих с аппаратным обеспечением напрямую или через процесс — посредник. В пункте 1.5 более подробно рассказывается о причинах связи языка Си и операционных систем.

Язык Си может использоваться для разработки программ для микроконтроллеров. Существует простое жизненное правило, чем меньше рюкзак, тем проще его перенести, главное, чтобы его содержимого хватило для решения всех насущных задач. Так и в программировании, чем меньше язык программирования, тем проще его перенести на другую платформу. В связи с этим ядро языка Си, т.е. набор ключевых слов очень мал. Под **ключевыми словами** понимаются символьные конструкции, значение которых зарезервировано стандартом языка. Зарезервированные слова не могут использоваться никаким другим способом, кроме как описанным в стандарте.

Перечислим ключевых слова языка Си:

auto, extern, short, while, break, float, signed, _Alignas, case, for, sizeof, _Alignof, char, goto, static, _Atomic, const, if, struct, _Bool, continue, inline, switch, _Complex, default, int, typedef, _Generic, do, long, union, _Imaginary, double, register, unsigned, _Noreturn, else, restrict, void, _Static_assert, enum, return, volatile, _Thread_local

Компактность языка достигнута с помощью включения в него только тех ключевых слов, которые соответствуют набору команд микропроцессора. Все остальные возможности вынесены в отдельные библиотеки. Любой другой подход не позволил бы использовать этот язык на микропроцессорах с ограниченным набором команд. Писать программы на Си можно даже для микропроцессоров, не имеющих собственных средств ввода/вывода, благодаря переносу всех функций стандартного ввода вывода в библиотеку стандартного ввода/вывода (**stdio - standart input output**), описываемую заголовочным файлов stdio.h.

Другими словами, язык Си очень ограничен во встроенных возможностях и не умеет даже получать и выводить информацию. Однако такая компактность позволяет изучать и использовать требуемые библиотеки по необходимости, что значительно ускоряет разработку программ.

Сложными темами при изучении языка Си являются: область видимости переменных, приведение типов, приоритет операторов, адресная арифметика, битовые операции, плохо спроектированные библиотеки. Также большие трудности создает отсутствие знаний в предметной области, часто это микроэлектроника, электроника, схемотехника, операционные системы. За исключением области видимости переменных, все остальные сложные темы можно изучать по мере

необходимости. Кроме того, существуют приемы, позволяющие существенно упростить работу с областями видимости переменных.

Любая запускаемая в ОС программа дополняет функции ОС и управляется ОС. Все современные ОС соответствуют стандарту операционных систем POSIX в той или иной степени. В соответствии с этим стандартом при запуске программы на исполнение, ОС создает три потока: поток ввода, поток вывода и поток ошибок. В терминологии языка Си поток ввода называется stdin, поток вывода stdout, поток ошибок stderr. Для программы, запускаемой в ЭТ, данные, получаемые с клавиатуры помещаются в поток ввода, данные, помещаемые в поток вывода, отображаются на экране, данные, помещаемые в поток ошибок, также отображаются на экране. Потоки можно перенаправлять и ассоциировать с различными устройствами и файлами. Обычно перенаправление потоков не требуется на первых этапах изучения языка программирования.

Программа на языке Си может состоять из одного файла, имя которого имеет расширение ***.c** . Звездочкой обозначается набор любых символов произвольной длинны. Расширение позволяет интегрированным средам разработки определять используемый язык программирования для выполнения подсветки ключевых слов, переменных и констант разных типов. Разработка языка велась в то время, когда объем доступной памяти ОЗУ не позволял загрузить команды программы в ОЗУ полностью. Было предложено выполнять деление исходного текста на два файла, один с расширением .c и второй с расширением ***.h**. Расширение ***.h** от английского head — т. е. файл заголовок или заголовочный файл. Таким образом существовала возможность писать программы, размер которых

превышал размер ОЗУ. Со временем, размер ОЗУ увеличился на порядки, но от использования заголовочных файлов все равно не отказались, т. к. они предоставляют значительные удобства для структурирования исходного текста программы с целью уменьшения времени повторной компиляции и применения концепции модульного программирования. Под модульным программированием подразумевается совместная работа нескольких программистов над одним проектом.

Сложность программы связана с объемом исходного текста программы и растет экспоненциально, т. е. программа 100 строк предположительно будет более чем в два раза сложнее, чем программа из 50 строк. При этом наиболее часто используют следующее деление:

- программы до 10000 строк считаются малыми;
- программы от 10000 до 100000 строк относятся к средним программам;
- программы свыше 100000 строк являются большими.

При создании средних, больших и сверхбольших программ, как правило, используется несколько файлов с расширением ***.c** и несколько файлов ***.h**. Соответствие между файлами программы и заголовочными файлами не всегда однозначное. Могут использоваться заголовочные файлы, не имеющие соответствующего файла программы, также наличие заголовочного файла для файла программы не является обязательным. Сведения о структуре исходных текстов на языке Си содержатся в стандарте языка программирования в разделе 5.1 Концептуальная модель стандарта языка.

Создание исполняемого файла из исходных текстов на языке Си происходит в три этапа:

1. Обработка исходных текстов предпроцессором.

2. Компиляция исходных текстов в объектный файл.

3. Сборка объектных файлов в исполняемый файл целевой операционной системы.

Рассмотрим этапы создания исполняемого файла на примере простейшего исполняемого файла. Обычно первая программа на новом языке программирования выводит всего одну фразу «Hello world!», обыгрывая первую публичную демонстрацию работы телефона передачей слова «Hello». В данном случае программа будет выводить текст «var =» и далее значение переменной var, хранимое в ОЗУ. Программа состоит из заголовочного файла main.h и файла программы main.c.

Заголовочный файл main.h

main.h

```
1  #include <stdio.h>
2  char var;
3
```

Файл содержит три строки. Первая строка является директивой предпроцессора. Директивы предпроцессора описаны в разделе 6.10 стандарта языка программирования. Предпроцессор обрабатывает первую строку следующим образом: он удаляет её из файла main.h и заменяет содержимым файла stdio.h. Файл stdio.h является заголовочным файлом стандартной библиотеки ввода-вывода, заключен в скобки из символов «<» и «>». Значит файл поставляется вместе с компилятором, расположен в каталоге /usr/include и содержит макросы, константы и прототипы функций, необходимые для использования функций стандартной библиотеки ввода/вывода, включая функции для работы с файлами. Вторая строка объявляет переменную типа char с идентификатором var. Идентификатор — это имя

переменной, константы или элемента сложного типа. Третья строка не содержит символов. В конце файла лучше оставлять пустую строку, не содержащую символов. Такое требование предъявлялось совсем старыми версиями компиляторов, но для совместимости, на всякий случай, лучше их ставить. Также они могут помочь при работе с файлами с исходными текстами программы, обработанными предпроцессором.

Файл программы main.c

main.c

```
1  #include "main.h"
2
3  int main () {
4      var = 0;
5      printf("var = %d\n", var);
6      return 0;
7  }
8
```

Файл main.c содержит 8 строк. Первая строка содержит директиву предпроцессора. Предпроцессор заменяет эту строку на содержимое файла main.h, однако файл должен находится в том же каталоге, что и файл main.c, т. к. в директиве его имя заключено в двойные кавычки.

Вторая строка не содержит символов и предназначена для визуального отделения блока объявлений от текста программы, обрабатываемого компилятором. Её наличие не обязательно.

В третьей строке определена главная функция программы main, с которой начинается исполнение всех программ на языке Си. Функция не принимает никаких аргументов, т. к. не содержит символов между скобками и после завершения своего исполнения возвращает значение типа целое число (int — от integer, т. е. целое

число). Тело, т. е. содержимое функции main, ограничено фигурными скобками «{» и в 7-й строке «}». Последняя 8-я строка также не содержит символов как предписывает раздел 5.1 Концептуальной модели стандарта языка Си. И хотя компиляторы обычно не замечают отсутствие в конце пустой строки, лучше её оставлять.

В четвертой строке записано выражение, которое переменной с идентификатором var присваивается численное значение, равное 0.

В пятой строке используется функция printf стандартной библиотеки ввода-вывода. В первом аргументе функции задается шаблон вывода вместе с форматом вывода переменной и служебная последовательность символов \n, обозначающая переход на новую строку, во втором идентификатор самой переменной.

В шестой строке записано ключевое слово return, позволяющее передать в вызывающую программу код возврата. Для данной программы вызывающей программой будет ЭТ.

Более подробное описание будет приводиться по мере разбора элементов языка.

Для создания исполняемого файла необходимо использовать ЭТ, описанный в пункте 2.3. Для этого необходимо выполнить щелчок правой кнопки мыши на рабочем столе или в любом свободном месте файлового менеджера и выбрать команду «Open terminal here», «Terminal» или «Терминал». Название команды будет меняться в зависимости от используемой платформы для рабочего стола.

В ЭТ необходимо выполнить команды:

cd ~

Сделать текущим домашний каталог пользователя.

mkdir source_code

Создать в домашнем каталоге пользователя каталог source_code.

cd source_code

Сделать текущим каталог source_code.

echo "#include <stdio.h>" > main.h

Вывести фразу "#include <stdio.h>" в поток вывода программы echo, который перенаправлен в файл main.h. Если файл main.h существует, его содержимое удаляется.

echo "char var;" >> main.h

Вывести фразу "char var;" в поток вывода программы echo, перенаправленный в файл main.h. Причем содержимое файла будет дополнено переданной фразой.

printf "\n" >> main.h

Вывести символ конца строки в поток вывода программы printf, перенаправленный в файл main.h. Причем содержимое файла будет дополнено.

Для проверки содержимого полученного файла используется команда

cat main.h

ЭТ с текущим каталогом source_code будет необходим для получения исполняемого файла на следующем этапе.

Для набора второго файла можно использовать редактор vim, описанный в пункте 2.4. Также можно использовать редакторы gedit, medit, leafpad, nano и другие. Для создания иллюстраций с исходным текстом программы использовалась среда разработки **Geany**.

После подготовки файлов с исходными текстами, необходимо преобразовать их с помощью компилятора в файл с исполняемыми командами. Для этого предлагается использовать компилятор **gcc**.

В ЭТ с текущим каталогом source_code необходимо выполнить команду

gcc main.c

Если компилятор не установлен, в ЭТ будет выведено

bash: gcc: команда не найдена

Для проверки наличия установленного компилятора выполняется команда

whereis gcc

При правильно установленном компиляторе будет выведено

gcc: /usr/bin/gcc /usr/lib/gcc /usr/lib64/gcc /usr/libexec/gcc /usr/share/man/man1/gcc.1.xz

При его отсутствии

gcc:

Для установки gcc необходимо повысить права доступа до уровня администратора ЭВМ с помощью команды

su -

затем необходимо ввести пароль администратора системы.

Получив права администратора системы, необходимо выполнить следующие команды:

apt-get update

обновить список версий, доступных для установки пакетов, включая список версий доступных ядер ОС.

update-kernel

обновить ядро операционной системы, т. е. Линукс. Если производится обновление ядра Линукс, для загрузки с новым ядром требуется перезагрузка ЭВМ.

apt-get dist-upgrade

установить новые версии уже установленных пакетов взамен старых.

Каждая команда работает в диалоговом режиме и требует ответов на ряд вопросов. Чаще всего программы запрашивают подтверждение на выполнение некоторого действия.

apt-get install gcc

установить пакет gcc.

Примерный вывод такой команды

```
[root@]# apt-get install gcc
Чтение списков пакетов... Завершено
Построение дерева зависимостей... Завершено
Виртуальный пакет gcc предоставляется следующими пакетами:
  gcc5 5.3.1-alt3 [Установлено]
  gcc4.9 4.9.2-alt5
  gcc4.8 4.8.2-alt5
  gcc4.7 4.7.2-alt10
  gcc4.6 4.6.3-alt11
  gcc4.5 4.5.4-alt4
  gcc4.4 4.4.7-alt4
  gcc4.3 4.3.2-alt20
  gcc4.1 4.1.2-alt12
  gcc3.4 3.4.5-alt16
Необходимо точно указать, какой из пакетов должен быть установлен.
E: Виртуальный пакет gcc предоставляется многими пакетами.

[root@]#
```

Вывод программы apt-get сообщает о том, что компилятор gcc5 версии 5.3.1 уже установлен.

В случае, если бы он был не установлен, для его установки потребовалась бы команда

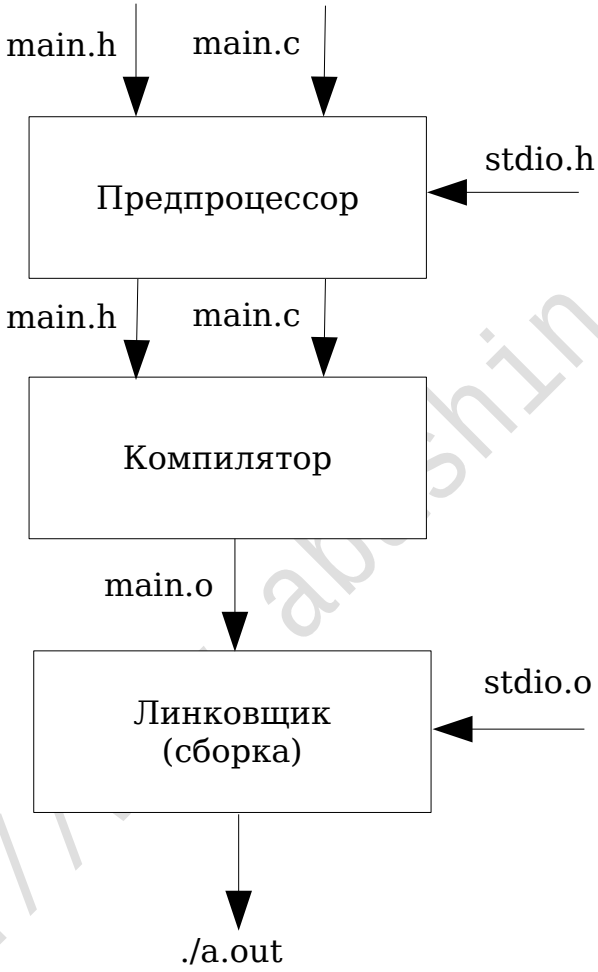
apt-get install gcc5

После установки компилятора вернемся к созданию исполняемого файла. В ЭТ с текущим каталогом source_code необходимо выполнить команду

gcc main.c

Программа gcc выполнит все этапы преобразования исходных текстов в исполняемый файл. Подробнее её

работа будет рассмотрена в пункте 5.2 этой книги. Представим упрощенную схему преобразования исходных текстов программы в исполняемый файл.



В результате будет получен исполняемый файл с именем a.out. Это имя является для программы gcc именем создаваемого исполняемого файла по умолчанию.

Если выполнить команду **ls** для каталога source_code, будет выведено имя исполняемого файла, созданного gcc.

Для запуска полученного файла используется команда

./a.out

Вывод в ЭТ

var = 0

В случае, если в исходном тексте содержится ошибка, компилятор предпринимает попытки её найти и выводит соответствующее сообщение. Например, если в файле main.c в строке 5 удалить символ «p», получится

rintf("var = %d\n", var);

Далее необходимо сохранить изменения в файл.

Выполнив команду gcc main.c, будет получен следующий вывод

```
[valerii@]$ gcc main.c
main.c: В функции «main»:
main.c:5:5: предупреждение: неявная декларация функции «rintf» [-Wimplicit-fu
nction-declaration]
  rintf("var = %d\n", var);
  ^
main.c:5:5: предупреждение: несовместимая неявная декларация внутренней функц
ии «rintf»
main.c:5:5: замечание: include «<math.h>» or provide a declaration of «rintf»
main.c:5:11: ошибка: несовместимый тип аргумента 1 функции «rintf»
  rintf("var = %d\n", var);
        ^
main.c:5:11: замечание: expected «float» but argument is of type «char *»
main.c:5:5: ошибка: слишком много аргументов в вызове функции «rintf»
  rintf("var = %d\n", var);
  ^
[valerii@]$
```

Каждая строка вывода программы gcc начинается с названия файла с исходным текстом, в котором обнаружена ошибка. Первое двоеточие отделяет номер строки с ошибкой, второе двоеточие отделяет номер символа, в котором обнаружена ошибка.

После удаления одного символа в названии функции библиотеки базового ввода/вывода printf и выполнения повторной компиляции, компилятор посчитал, что используется математическая функция rintf и вывел 2 предупреждения, 2 замечания и 2 ошибки. Однако в данном случае он смог точно указать место начала проблемы. Это символ 5 строки 5 файла main.c.

Приведенный пример демонстрирует главное правило чтения вывода сообщений компилятора. Их необходимо читать сверху вниз и по порядку. Не следует забегать вперед, пока не исправлены ошибки выше. Часто исправление одной ошибки приводит к исправлению нескольких последующих.

3.2. Типы данных. Операторы ввода/вывода

Основные принципы построения ОС описаны в пункте 1.5. Одним из них является принцип простой метафорики, состоящей всего из двух понятий: вычислительный процесс и файл. Под файлами в первую очередь понимается способ хранения данных. В языке Си для хранения данных используются переменные и константы, описанные одним из типов данных, которые могут храниться только в ОЗУ. Обработка данных в регистрах контроллера или микропроцессора скрыта конструкциями языка Си. Данные, хранимые на носителях информации, устройствах или в вычислительной сети, доступны с помощью функций внешних библиотек.

Назначение типов данных — это помощь в нахождении ошибок компилятором в исходном тексте программы. Соответственно на этапе выполнения исполняемых команд типов данных не существует, т.к. микропроцессор умеет выполнять операции только с данными, размер которых равен размеру машинного слова микропроцессора или кратен ему.

Каждое значение или набор значений в языке Си является некоторой последовательностью ячеек ОЗУ, которые логически связаны между собой. Имеется ввиду тот факт, что теоретически один элемент массива может физически располагаться на одной физической планке ОЗУ, а следующий на другой. Однако ОС предоставляет некоторую абстракцию, в рамках которой обе планки памяти являются единым адресным пространством с возрастающим адресом для каждой последующей ячейки.

Все данные в языке Си делятся на несколько категорий.

1. Переменная, константа или указатель. Константа отличается от переменной тем, что её значение задается один раз и не может быть изменено до завершения выполнения программы. Значение переменной может меняться неограниченное количество раз. Указатель может содержать адрес как переменной или константы любого типа, так и адрес элемента составного типа, о которых будет рассказано далее.

2. Значение может быть знаковым или беззнаковым. Первый бит знаковой переменной или константы используется для определения знака числа, положительное оно или отрицательное. Беззнаковая переменная может вместить большее положительное значение.

Каждая переменная, константа или указатель описывается идентификатором (именем переменной), имеет свой адрес, соответствующий адресу первой ячейки памяти, начиная с которой располагаются её значения, и занимает некоторое количество ячеек ОЗУ. Тип данных, в свою очередь, определяет допустимый диапазон значений для соответствующей переменной или константы и допустимые для неё операции.

В языке Си существует набор типов, называемых сложными или составными типами данных. Основное их отличие — в использовании одного идентификатора для нескольких элементов, каждый из которых по-сути является отдельной переменной или константой. Подробнее составные типы будут рассмотрены в следующих разделах.

Стандарт языка Си в пункте 6.2.5. Типы, содержит описание допустимых типов данных. Спецификаторы типов, т.е. идентификаторы, которыми задаются их названия, описаны в пункте 6.7.2. Спецификаторы типов. Можно выделить несколько категорий типов данных:

1. Архитектурнозависимые целые типы данных, размер которых может меняться в зависимости от аппаратной платформы. К ним относятся: **char**, **short int**, **int**, **long int**, **long long int**.

2. Целочисленные типы с минимальнодопустимой или фиксированной разрядностью. Например: **int_least8_t**, **uint_least8_t**, **int_least16_t**, **uint_least16_t**, **int_least32_t**, **uint_least32_t**, **int_least64_t**, **uint_least64_t**; это типы с минимально допустимым размером объема памяти для значения типа. **int*N*_t**, где *N* = 8, 16, 32, 64, фиксированный объем памяти для этого типа.

3. Числа с плавающей запятой: **float**, **double**, **long doube**.

4. Комплексные и мнимые числа: **float _Complex**, **double _Complex**, **long doube _Complex, _Imaginary**.

5. Указатели, размер которых зависит от типа переменной или константы, на который они указывают.

6. Составные типы: массивы, структуры, объединения, битовые маски.

7. Типы, предлагаемые разработчиками компиляторов языка Си.

Существуют также рекомендуемые типы данных. Во избежания возникновения путаницы, рекомендуется использовать только те типы данных, однозначное описание которых приведено в стандарте языка программирования. На практике, все потребности программиста при разработке программы удовлетворяет не более 10 типов, остальные нужны только для чтения исходных текстов, написанных другими программистами.

Для обработки текстовых значений в языке используются массивы типов char или **wchar** для мультибайтовых кодировок. Массивы будут рассмотрены в следующих разделах. Специальные типы для работы с

текстом рассматриваются при работе над новым стандартом.

Максимальные и минимальные значения допустимые для переменных и констант разных типов и описаны в разделе 5.2.4.2. Пределы значений стандарта языка. Стандарт предписывает поставлять вместе с компилятором файл **limits.h**, в котором также приведены допустимые диапазоны для разных типов. При использовании компилятора gcc этот файл можно найти в каталоге **/usr/include/** .

При изучении типов данных необходимо также обратить внимание на заголовочные файлы **stdint.h**, **float.h**, **stdbool.h**. Следует помнить, что разработчики компилятора gcc могут менять расположение этих заголовочных файлов или отказаться от них, переместив их содержимое в другие заголовочные файлы.

Для использования каждой константы, переменной или указателя необходимо обязательно выполнить его **объявление**, т. е. задание идентификатора. Также крайне рекомендуется выполнять **инициализацию** переменных, т. е. задание начального значение. Правила инициализации, немного отличные от других операций для составных типов данных, и будут рассмотрены в соответствующих разделах.

На рисунке 1 приведен исходный текст программы, демонстрирующей объявления переменных, константы и указателей разных типов. Реализация главной функции main начинается со строки 3. Тело функции открывается открывающей фигурной скобкой «{».

В строке 4 объявлена константа типа char с идентификатором ch1, которая инициализирована кодом символа «a».

В строке 5 объявлена переменная типа short int с идентификатором shr. При объявлении типа short int,

достаточно использовать ключевое слово short. Переменная инициализирована значением 15 в десятичном виде.

В строке 6 объявлена переменная типа int с идентификатором dig, которая инициализирована значением 4.

В строке 7 объявлена переменная ch2, типа беззнаковый char и инициализирована числом 1.

main.c

```
1  #include <stdio.h>
2
3  int main () {
4      const char ch1 = 'a';
5      short shr = 15;
6      int dig = 4;
7      unsigned char ch2 = 1;
8      printf("ch1 = %c, shr = %d, dig = %d, ch2 = %d\n",
9                                          ch1, shr, dig, ch2);
10     float fl = 3.0;
11     float * fl_p = &fl;
12     double * dbl = NULL;
13     printf("fl = %2.1f, fl_p = %p, fl_p* = %f, dbl = %p\n",
14                                          fl, fl_p, *fl_p, dbl);
15     return 0;
16 }
17
```

Рисунок 1 — Исходный текст программы

В строках 8-9 записана функция форматированного вывода printf стандартной библиотеки ввода/вывода. В шаблоне вывода указана строка, в которую подставлены спецификаторы типов для выводимых аргументов. Выводимыми аргументами являются идентификаторы переменных.

В строке 10 объявлена переменная с идентификатором fl типа с плавающей запятой float, инициированная значением 3.0. При задании констант с плавающей запятой, необходимо указывать дробную часть через точку, даже если она равна 0.

В строке 11 объявлен указатель на переменную типа float с идентификатором fl_p, который инициирован результатом выполнения операции взятия адреса переменной fl. Операция взятия адреса обозначается с помощью унарного оператора **&**.

В строке 12 объявлен указатель на переменную типа с плавающей запятой удвоенного размера (тип double) с идентификатором dbl, которая инициирована значением NULL. **NULL** — означает указатель в никуда, определяется с помощью заголовочного файла, поставляемого с компилятором и обычно равен 0, однако заменять его 0 не рекомендуется.

В строке 13-14 записан вызов функции форматированного ввода/вывода, который выводит первую переменную в формате две цифры до запятой и одна цифра после запятой. Вместо второй переменной выводится её адрес. Третья переменная выводится как переменная с плавающей запятой. Четвертая переменная выводит значение адреса. Кроме того, к переменной fl_p в четвертом аргументе применена операция разыменования, которая обозначается звездочкой «*».

В строке 15 записано ключевое слово, задающее значение, возвращаемое в ЭТ при завершении работы программы.

В строке 16 записан один символ, закрывающий тело функции main.

На рисунке 2 представлены команда компиляции исходного текста программы, его исполнения и вывод программы в стандартный поток вывода. Все команды выполняются после задания текущим каталога содержащего файл main.c!

```
[valerii@]$ gcc main.c
[valerii@]$ ./a.out
ch1 = a, shr = 15, dig = 4, ch2 = 1
fl = 3.0, fl_p = 0x7ffc0b783548, fl_p* = 3.000000, dbl = (nil)
[valerii@]$
```

Рисунок 2 — Вывод программы в ЭТ

Функция форматированного вывода строк 8-9 сформировала первую строку вывода, в которой в текст шаблона вывода, вместо спецификаторов типов вставлены значения инициированных переменных.

Функция, записанная в строках 13-14, сформировала вторую строку вывода. Переменная fl выведена с шаблоном одна цифра две цифры до запятой и одна после. Переменная fl_p является указателем и вывела адрес, по которому располагается переменная fl. Разыменование переменной fl_p с помощью операции * привело к выводу значения переменной fl. Однако т. к. формат не указан, после запятой было выведено значение символов по умолчанию, а именно 6 нулей. Переменная dbl инициирована значением NULL, которое обозначено функцией printf как «(nul)».

Более подробная информация о типах содержится в соответствующих разделах стандарта языка программирования.

Рассмотрим основные операторы ввода/вывода. Для их использования необходимо подключать заголовочный файл stdio.h. Наиболее простыми и быстрыми функциями являются gets и puts, потому что их использование не требует преобразования аргументов в другой формат. В соответствии со стандартом языка Си, каждая функция должна быть реализована до своего первого вызова или иметь прототип, который по тексту встречается раньше её вызова. Подробно правила написания функций будут

рассмотрены в соответствующем пункте.

Прототип функции gets описан в стандарте следующим образом

*char *gets(char *s);*

Функция gets читает символы из стандартного потока ввода stdin и размещает считанные данные в массив s, указатель на который передан в качестве аргумента функции, пока не встретит символ конца файла или символ конца строки. Завершающий символ отбрасывается, а вместо него записывается null, обычно соответствующее значению 0.

Если функция завершилась успешно, она возвращает указатель на массив, переданный её в качестве аргумента. Подробнее о массивах будет рассказано в последующих пунктах. Если функция завершилась неудачей, возвращается значение null.

Полное описание функции содержится в пункте 7.19.7.7 стандарта.

В пункте 7.19.7.10 содержится описание функции puts. Данная функция имеет прототип

*int puts(const char *s);*

и выводит символы, содержащиеся по адресу s, в стандартный поток вывода stdout и возвращает количество выведенных символов. Тип, передаваемый в функцию puts, соответствует константному значения, а значит изменение массива символов внутри функции puts невозможно.

В разделе 7.19.6 описаны функции форматированного ввода/вывода. Функция форматированного вывода fprintf описана в пункте

7.19.6.1 и имеет прототип

*int fprintf(FILE * restrict stream, const char * restrict format, ...);*

Функция принимает следующие аргументы:

- restrict stream — указатель на структуру типа FILE, которая может быть как stdin, stdout, stderr, так и указателем на файл на накопителе или на файл-устройство;

- restrict format — шаблон вывода, являющийся строкой со вставленными спецификаторами типов, вместо которых будут подставлены значения переменных;

- ... — означает возможность использовать любое количество аргументов, которые будут подставлены вместо спецификаторов типов выводимых переменных.

Допустимые спецификаторы типов описаны в том же пункте стандарта, что и сама функция.

Часто используется сокращенная форма записи оператора fprintf. Она отличается отсутствием возможности указания потока вывода и всегда использует стандартный поток вывода stdout. Функция называется printf, описана в пункте 7.19.6.3 и имеет прототип

*int printf(const char * restrict format, ...);*

Функция форматированного ввода fscanf является «обратной» fprintf, в том смысле, что её использование схоже с применением fprintf, только используется для получения данных из стандартного потока ввода. Функция fscanf описана в пункте 7.19.6.2. и имеет прототип

*int fscanf(FILE * restrict stream, const char * restrict format, ...);*

Функция также как и fprintf использует шаблон ввода, в котором указываются спецификаторы типов. Сокращенная форма записи имеет прототип

*int scanf(const char * restrict format, ...);*

и описана в пункте 7.19.6.4.

inout.c

```c
#include <stdio.h>

int main () {
    char buffer[100] = {0};
    int dig = 0;

    puts("Введите слово");
    gets(buffer);
    puts((char *)&buffer[0]);

    puts("Введите цифру");
    fscanf(stdin,"%d",&dig);
    fprintf(stdout,"Введена цифра: %d\n", dig);

    puts("Введите цифру");
    scanf("%d",&dig);
    printf("Введена цифра: %d\n", dig);

    return 0;
}
```

Рисунок 3 — Исходный текст программы примера работы операторов ввода/вывода

Приведем пример использования рассмотренных функций ввода / вывода с помощью исходного текста, набранного в файл inout.c. Назначение первой, второй и

третьей строки подробно описано в предыдущем пункте. В строке 4 производится инициализация переменной с идентификатором buffer, которая является массивом из 100 элементов. Подробно массивы будут рассмотрены в следующих разделах.

В строке 5 объявляется знаковая переменная целого типа с идентификатором dig и инициализируется значением 0.

Строки 6, 10, 14, 18 оставлены пустыми для наглядности. Пустая строка 21 добавлена для совместимости со старыми версиями компиляторов.

Строка 7 состоит из вызова функции puts. Скобками ограничен аргумент, передаваемый в функцию. В этой строке в функцию передается константное значение «Введите слово», которому не присвоен идентификатор. Вызов функции заканчивается точкой с запятой, т. к. стандарт языка Си требует завершать этим символом каждое выражение.

Строка 8 содержит вызов функции gets, который принимает аргументом указатель на массив с идентификатором buffer. При вызове этого оператора, выполнение программы будет приостановлено и будет отображаться приглашение для ввода слова пользователем в виде прямоугольника, который будет заменен введенным символом. Для завершения ввода необходимо нажать клавишу Enter. После этого исполнение программы продолжится.

Строка 9 содержит вызов функции puts, но в этот раз в неё передается указатель на первый элемент массива buffer, т. к. цифра 0 в квадратных скобках обозначает не порядковый номер элемента, а на сколько элементов нужно сместиться в памяти процесса от начала массива.

Строка 11 отличается от строки 7 только константой, передаваемой в функцию puts.

Строка 12 содержит вызов функции fscanf. В него передается три аргумента. Первый аргумент stdin — это стандартный поток ввода, ассоциированный с клавиатурой. Второй аргумент — это шаблон ввода, содержащий только один формат переменной целого типа. Третий аргумент состоит из унарной, т. е. содержащей только один операнд, операции взятия адреса и идентификатора знаковой переменной целого типа. Таким образом, третьим аргументом будет константа типа адрес знаковой переменной целого типа. В результате вызова этой функции данные будут читаться из стандартного потока ввода до нажатия клавиши Enter, затем они будут преобразованы в целое число и записаны в область ОЗУ, адресуемую идентификатором dig.

Строка 13 содержит вызов функции fprintf, первый аргумент которой stdout соответствует стандартному потоку вывода. Второй аргумент соответствует шаблону вывода. Символы данного шаблона вывода будут выведены на экран, за исключением последовательности %d, которая будет заменена на значение переменной с идентификатором dig, и последовательности \n , которая означает переход на новую строку.

Строка 15 идентична строке 11.

Строка 16 содержит вызов оператора scanf, аргументы которого отличаются от вызова в строке 12 только отсутствием указания потока ввода.

Строка 17 содержит вызов оператора printf, который совпадает с вызовом строки 13, за исключением указания потока вывода.

Строка 19 содержит ключевое слово return, которое передает значение 0 в вызывающую программу, т.е. ЭТ.

Для создания программы необходимо запустить ЭТ, сделать каталог с исходным текстом программы текущим и выполнить команду, указанную на рисунке 4. Команда

состоит из двух, объединенных логическим оператором И (&&). Первая часть команды gcc inout.c -o inout предписывает выполнить создание исполняемого файла программы из исходного текста на языке Си, расположенного в файле inout.c, а исполняемый файл назвать inout без расширения. Вторая часть команды ./inout предписывает запустить полученный исполняемый файл на исполнение.

```
[valerii@]$ gcc inout.c -o inout && ./inout
inout.c: В функции «main»:
inout.c:8:5: предупреждение: неявная декларация функции «gets» [-Wimplicit-funct
ion-declaration]
     gets(buffer);
     ^
/tmp/.private/valerii/ccyOz2bW.o: In function `main':
inout.c:(.text+0x52): warning: the `gets' function is dangerous and should not b
e used.
Введите слово
Программа
Программа
Введите цифру
1
Введена цифра: 1
Введите цифру
1
Введена цифра: 1
[valerii@]$
```

Рисунок 4 — Создание и исполнение программы примера использования функций форматированного ввода/вывода

При создании исполняемого файла из исходного текста программы были получены следующие сообщения:

inout.c: В функции «main»:

В файле inout.c в функции с идентификатором main.

inout.c:8:5: предупреждение: неявная декларация функции «gets» [-Wimplicit-function-declaration]
gets(buffer);
^

В файле inout.c в строке 8 символе 5 предупреждение об использовании неявной декларации функции gets, появление которой в данном случае, связано с необходимостью преобразования входного аргумента к типу указатель на тип char.

/tmp/.private/valerii/cczPEvD3.o: In function `main':
inout.c:(.text+0x52): warning: the `gets' function is dangerous and should not be used.

Данное сообщение предупреждает, что использование функции gets опасно и от неё будет произведен отказ в следующих стандартах языка программирования.

Последующий вывод, начиная с фразы «Введите слово», сгенерирован программой inout, созданной из рассмотренного исходного текста программы. Из примера следует, что использование операторов fscanf и fprintf с указанием стандартных потоков ввода и вывода соответственно ничем не отличается от использования операторов printf и scanf.

3.3. Выражения. Арифметические и логические операции. Математические функции

Выражение — это последовательность операторов, операндов, других объявленных объектов или вызовов функций, которые допустимы в рамках языка программирования. Обычно выражения отделяются друг от друга с помощью точки с запятой, при условии что оно не расположено внутри скобок, квадратных скобок, не является аргументом функции. Полное определение понятия выражения дано в пункте 6.5 стандарта языка Си.

Примерами выражения являются:

```
i = j + 2;
printf("%d", k);
j - t && t + j
```

Бинарные арифметические выражения записываются следующим образом: сначала записывается идентификатор, по которому будет присвоено значение, полученное в результате вычисления выражения, далее ставится знак присвоения = , после знака присвоения записывается вычисляемая часть выражения. Под бинарными арифметическими выражениями подразумеваются те, в которых операция выполняется над двумя значениями, например 3+5. Действия, выполняемые над одним значением, например отрицание единицы, обозначаемое восклицательным знаком !1, которое будет равно 0, называется унарной операцией.

Примеры арифметических операций:

```
x = y + 2;
```

```
x = y — 2;
x = y * 2;
x = y / 2;
```

Для выполнения арифметической операции с одной переменной возможно использование короткой записи выражения, например:

```
int x = 2;
x += 2;
```

полностью соответствует записи

```
x = x + 2;
```

в первом и во втором случае x будет равно 4.

Другой пример:

```
int x = 8;
x /= 2;
```

Переменная x будет равна 4.

Особенностью языка Си является отбрасывание дробной части, если результат деления присваивается целочисленной переменной. Для получения остатка от деления используется специальный оператор. Например:

```
int x = 9;
int y = x / 2;
```

В результате вычисления выражения, переменная y будет инициализирована целочисленным значением 4. Остаток от деления будет отброшен.

Для получения остатка от деления используется

оператор %. Например:

```
int x = 9;
int y = x % 2;
```

Переменная y будет равна 1. Более подробно этот вопрос будет рассмотрен в пункте, посвященном преобразованию типов данных.

Арифметические действия можно записывать в виде выражений с использованием скобок, например:

```
int y = (x — 3) / (x — 2);
```

при этом приоритет выполнения операция будет соответствовать правилам арифметики.

В языке Си в качестве знака присвоения используется символ, обозначающий равенство в математике. Это вызывает путаницу на начальных этапах изучения языка. Математическое равенство в языке Си обозначается как логическая операция сравнения и обозначается двумя символами == . Предлагается следующий способ запомнить отличия. Считать, что знак присвоения эквивалентен действию положить яблоко в корзинку, т. е. поместить некоторое значение в ячейки памяти. Знак сравнения содержит два символа и поэтому предлагается ассоциировать его с весами, на которые кладут два яблока и выполняют сравнение, какое из них больше.

Операции сравнения, проверка на равенство, проверка на неравенство и другие операции сравнения приведены в пункте 6.5.9 стандарта Си.

Для рассмотрения логических, побитовых операций и операций сдвига обратимся к исходному тексту программы, представленному на рисунке 1.

```
01.c
1  #include <stdio.h>
2
3  int main ()
4  {
5      int j = 3;
6      int t = 1;
7      printf("%d=%d+%d\n", j+t, j, t);
8      printf("%d=%d&%d\n", j&t, j, t);
9      printf("%d=%d&&%d\n", j&&t, j, t);
10     printf("%d=%d|%d\n", j|t, j, t);
11     printf("%d=%d||%d\n", j||t, j, t);
12     printf("%d=~%d %d=~%d\n", j, ~j, t, ~t);
13     printf("%d=!%d %d=!%d\n", j, !j, t, !t);
14     printf("%d=%d<<%d\n", j<<t, j, t);
15     printf("%d=%d>>%d\n", j>>t, j, t);
16     return 0;
17 }
18
```

Рисунок 1 — Выполнение логических, побитовых операций и операций сдвига

Строки 1-4, 16-18 подробно описаны в пункте 3.1.

Строка 5 объявляет переменную целого типа с идентификатором j и инициализирует её константой 3.

Строка 6 объявляет переменную целого типа с идентификатором t и инициализирует её константой 1.

Строки 7-14 состоят из выражений, содержащих один вызов оператора printf, и отличаются только передаваемыми в него аргументами. В строке 7 первый аргумент является шаблоном вывода "%d=%d+%d\n". Спецификаторы типа %d будут заменены следующим образом: первый спецификатор будет замен результатом вычисления выражения j+t. В конце выражения не ставиться точка с запятой, т. к. выражение является аргументом оператора printf. Второй будет заменен значением переменной с идентификатором j. Третий заменен значением переменной с идентификатором t.

Замечание. Обычно результат вычисления выражения записывается в ячейки, соответствующие некоторой переменной. Эти действия соответствуют двум действиям микропроцессора: выполнить сложение и поместить результат в регистр микропроцессора eax, поместить результат в переменную с указанным идентификатором в ОЗУ, т. к. все переменные, по определению, расположены в ОЗУ. В строке 7 результат выражения j+t вычисляется в микропроцессоре и сохраняется в регистре микропроцессора, например, в eax. Далее микропроцессор размещает другие аргументы в своих других регистрах и переходит к выполнению операторов функции printf. Таким образом создается текст программы, которая будет выполняться быстрее. Для более глубокого понимая таких механизмов необходимы знания в области микропроцессорной техники и программировании на ассемблере.

В строке 8 операция сложения заменена бинарной операцией побитового И (&). В результате выполнения этой операции будут выполнены операции И между соответствующими разрядами переменных j и t, и если оба разряда будут равны 1, в соответствующий разряд регистра eax будет помещена единица. После выполнения операций со всеми разрядами получится единое многоразрядное значение, которое будет выведено на экране оператором printf.

В строке 9 выполняется бинарная операция логическое И (&&). Результатом выполнения этой операции является 1, если оба операнда не равны 0, в противном случае результат будет равен 0.

В строке 10 выполняется бинарная операция — побитовое ИЛИ (|). Эта операция выполняется так же, как и операция побитового И в строке 8, за тем исключением, что для получения значения 1 в

результирующем разряде необходимо, чтобы хотя бы один из двух соответствующих разрядов входных операндов был равен 1. В ином случае, значение результирующего разряда будет равно 0.

В строке 11 выполняется бинарная операция — логическое ИЛИ (||). Результатом этой операции будет единица, если хотя бы один из операндов будет отличен от нуля. В противном случае он будет равен 0.

В строке 12 выполняется унарная операция побитового отрицания (~). В результате выполнения этой операции каждый бит операнда будет изменен на противоположный. При использовании знакового типа целого числа, первый бит обозначает знак числа. Его изменение приведет к интерпретации числа как отрицательного, т. е. записанного в дополнительном виде, см. пункт 1.2.

В строке 13 выполняется унарная операция логического отрицания (!). Эта операция возвращает 0, если его операнд отличен от 0 или 1, если его операнд равен 0.

В строке 14 выполняется бинарная операция сдвига влево (<<). Арифметически сдвиг влево на 1 соответствует умножению на 2. Данная операция заключается в том, что каждый бит числа будет перезаписан на одну позицию левее, а в освободившийся разряд будет помещен 0.

В строке 15 выполняется бинарная операция сдвига вправо (>>). Арифметически сдвиг вправо на 1 соответствует делению на 2. Данная операция заключается в том, что каждый бит числа будет перезаписан на одну позицию правее, а в освободившийся разряд будет помещен 0. Крайний правый бит будет потерян.

Приоритет арифметических, логических и побитовых

операций сведен в таблицу в разделе 2.12 Книги «Язык программирования Си» Брайана Кернигана и Денниса Ритчи.

В стандарте языка программирования предусмотрены альтернативные названия логических операций, описанные в разделе 7.9 Альтернативные обозначения, которые объявлены в файле iso646.h.

Считается, что операции сдвига позволяют ускорить исполнение программы, однако это справедливо только, если операции сдвига составляют большую часть вычислительной сложности исполняемого файла.

```
[valerii@]$ gcc 01.c && ./a.out
4=3+1
1=3&1
1=3&&1
3=3|1
1=3||1
3=~-4 1=~-2
3=!0 1=!0
6=3<<1
1=3>>1
[valerii@]$
```

Рисунок 2 — Результат исполнения исходного текста представленного на рисунке 1

Для запуска описанного исходного текста программы используется команда ЭТ, содержащая логическое И причем вторая часть — запуск полученного исполняемого файла, производится только в случае успешного завершения команды создания исполняемого файла скриптом gcc.

Операции сравнения будут подробно рассмотрены в пункте, посвященном условным переходам.

Для выполнения математических операций,

отличных от арифметических, необходимо выполнять подключение модуля math. Для этого исходный текст программы должен содержать строку

#include <math.h>

При компиляции исходных текстов, содержащих математические функции модуля math, необходимо указывать ключ -lm, например так:

gcc 02.c -o 02 -lm

Все библиотеки кроме стандартных (stdio.h, stdlib.h), которые подключены по умолчанию, требуют для компоновки указания специального ключа.

Математические функции модуля math описаны в пункте 7.12 Математика стандарта языка Си. Наиболее часто используемыми являются функции возведения в степень, извлечения корня и тригонометрические функции.

Пример использования математических функций представлен на рисунке 3.

Поясним значение некоторых строк текста программы, представленной на рисунке 3.

В строке 2 указана директива предпроцессора. Предпроцессор обрабатывает эту строку, заменяя её на содержимое файла math.h, расположенного в каталоге include, поставляемой вместе с компилятором. Добавление этой строки позволяет использовать математические функции, прототипы которых описаны в файле math.h.

02.c

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

int main ()
{
    double xd = 0.356;
    float xf = 0.356;
    double yd = 0.322;

    printf("арккосинус %e=acos(%3.5e)\n", acos(xd), xd);
    printf("арксинус %f=asinf(%f)\n", asinf(xf), xf);
    printf("арктангенс %e=atan(%e)\n", atan(xd), xd);
    printf("арктангенс y/x %e=atan2(%e)\n", atan2(yd, xd), xd);
    printf("косинус %f=cosf(%f)\n", cosf(xf), xf);
    printf("синус %e=sin(%e)\n", sin(xd), xd);
    printf("тангенс %f=tanf(%f)\n", tanf(xf), xf);
    printf("арк гиперболический косинус %f=acoshf(%f)\n", acoshf(xf), xf);

    printf("экспонента 2^x %e=exp2(%e)\n", exp2(xd), xd);
    printf("экспонента e^x %e=exp(%e)\n", exp(xd), xd);

    printf("натур.логарифм %f=logf(%f)\n", logf(xf), xf);
    printf("логарифм по осн.10 %e=log10(%e)\n", log10(xd), xd);

    printf("x^y %e=pow(%e,%e)\n", pow(xd, yd), xd, yd);
    printf("кв.корень %f=sqrtf(%f)\n", sqrtf(xf), xf);

    printf("ближ.целое меньше %e %e=floor(%e)\n", xd, floor(xd), xd);

    return 0;
}
```

Рисунок 3 — Исходный текст программы использующей математические функции

В строке 6 объявляется и инициализируется переменная с идентификатором xd, типа double. Тип double обозначает переменные и константы, хранящие значение в виде мантисы и степени удвоенной точности, т. е. значение с плавающей запятой удвоенной точности. После знака присвоения указана константа с дробной частью. Необходимо запомнить, при инициализации

переменных и констант с плавающей запятой всегда необходимо указывать дробную часть, даже если она равна 0. Под объявлением понимается, что будет выделена память в ОЗУ для хранения переменной, а под инициализацией понимается присвоение начального значения переменной.

Строка 7 также объявляет и инициализирует переменную с плавающей запятой.

```
[valerii@]$ gcc 02.c -lm && ./a.out
арккосинус 1.206812e+00=acos(3.56000e-01)
арксинус 0.363984=asinf(0.356000)
арктангенс 3.420100e-01=atan(3.560000e-01)
арктангенс y/x 7.352926e-01=atan2(3.560000e-01)
косинус 0.937298=cosf(0.356000)
синус 3.485278e-01=sin(3.560000e-01)
тангенс 0.371843=tanf(0.356000)
арк гиперболический косинус -nan=acoshf(0.356000)
экспонента 2^x 1.279872e+00=exp2(3.560000e-01)
экспонента e^x 1.427608e+00=exp(3.560000e-01)
натур.логарифм -1.032825=logf(0.356000)
логарифм по осн.10 -4.485500e-01=log10(3.560000e-01)
x^y 7.170788e-01=pow(3.560000e-01,3.220000e-01)
кв.корень 0.596657=sqrtf(0.356000)
ближ.целое меньше 3.560000e-01 0.000000e+00=floor(3.560000e-01)
[valerii@]$
```

Рисунок 4 — Результат исполнения исходного текста, представленного на рисунке 3

В строке 10 производится вызов оператора printf. Первый аргумент этого оператора является шаблоном вывода. Он будет выведен на экран за исключением последовательностей %e, которые являются спецификаторами типа double оператора printf. Первый спецификатор типа будет заменен результатом вызова математической функции acos, второй — значением переменной с идентификатором xd. Арккосинус вычисляется от одного значения, т. е. координаты по оси

x, поэтому в функцию acos также передается одно значение с идентификатором xd. В конце шаблона вывода используется последовательность символов «/n», обозначающая переход на новую строку.

В строке 13 используется функция вычисления аргтангенса y/x, поэтому в функцию atan2 передается два аргумента с идентификаторами xd, yd.

Для создания программы из исходного текста программы и её запуска в файле 02.c используется команда, представленная в первой строке на рисунке 4. Команда состоит из двух частей. Первая часть вызывает компилятор gcc и передает в него в качестве аргументов имя файла с исходным текстом программы и ключом -lm. Программа gcc передает ключ -lm компоновщику ld, который включает в создаваемый исполняемый файл объектный файл библиотеки math.

Следует обратить внимание на то, что аргументы приведенных математических функций, т. е. передаваемые им значения, должны не противоречить математическим правилам.

3.4. Операторы выбора

Наиболее популярный оператор условного перехода if описан в стандарте языка Си в пункте 6.8.4.1 и имеет 2 формы записи.

Полная форма:

if (выражение) { операторы }
else { операторы };

Сокращенная форма записи:

if (выражение) { операторы };

Если после ключевого слова if и скобки используется один оператор, фигурные скобки можно не использовать.

if (выражение) оператор;

Также не использовать фигурные скобки можно только в одной части полного оператора if.

if (выражение) оператор;
else { операторы };

Есть простое правило, которое позволяет не запутаться с необходимостью выставления фигурных скобок. Их нужно ставить всегда. С появлением нужного опыта этот вопрос отпадает сам собой.

Следует иметь в виду, что результатом выражения является логическое значение, соответствующее одному из двух значений — истина или ложь. Приведем примеры задач, которые можно решить с использованием этих ключевых слов.

Условие задачи:

Если значение переменной с идентификатором left равно 1, необходимо вызвать функцию поворота налево.

Решение:

```
if ( left == 1 ) move_left();
```

Условие задачи:

Если значение переменной с идентификатором left равно 2, вызвать функцию поворота налево и вызвать функцию остановки.

Решение:

```
if ( left == 1 ) {
      move_left();
      stop ();
};
```

Условие задачи:

Если значение переменной с идентификатором right равно 1, вызвать функцию поворота направо, иначе повернуть налево и остановиться.

Решение:

```
if ( right == 1 ) move_right();
else {
      move_left();
      stop ();
};
```

Условие задачи:

Если значение переменной с идентификатором right равно 1 и значение переменной left равно 0, вызвать функцию поворота направо.

Решение:

```
if ( right == 1 && left == 0 ) move_right();
else {
      move_left();
      stop ();
};
```

Пояснение:

В данном примере выражение в скобках после ключевого слова if содержит две операции сравнения, объединенные операцией логического И. Приоритет логической операции И меньше, чем у операции сравнения, поэтому сначала будет произведено сравнение переменной с идентификатором right с константой 1. Второй шаг будет выполнен только тогда, если результатом первого сравнения будет 1, т.е. истина. Вторым шагом станет сравнение переменной с идентификатором left с константой 0. Третий шаг — выполнение логической операции И над первым результатом, равным 1 и результатом второй операции. Логическая И возвращает 1 в случае, если оба значения равны 1, во всех других случаях она вернет 0 и будут выполнены операторы в фигурных скобках после ключевого слова else.

Для упрощения понимания выражения следует использовать скобки, руководствуясь правилами арифметики. Предыдущее решение можно записать так:

```
if ( (right == 1) && (left == 0) )
```

Добавление скобок упрощает восприятие условия. Если условие содержит более 3 операций, скорее всего его разбиение на несколько операторов if ускорит выполнение программы. Для этого необходимо вынести в первый оператор условие, которое будет встречаться

реже.

```
if ( right == 1 )
    if ( left == 0 ) move_right();
else {
    move_left();
    stop ();
};
```

Предположим, что в программе значение right чаще равно 0. В таком случае проверка left равно 0 будет производиться только, если right равно 1. В данном случае это не дает прироста скорости исполнения, т.к. второе выражение и так не будет выполнено, если первое равно ложь. Причина отсутствия ускорения программы в том, что в выражении всего 3 операции.

Существует форма операции выбора без использования ключевого слова if. Его форма:

условие ? оператор : оператор;

Представленная форма оператора выбора обычно используется в качестве аргумента при вызове некоторой функции.

Например:

```
int var = 0;
printf ("&s", var?"да":"нет");
```

Программа, созданная из этого исходного текста программы, всегда будет выводить

нет

Первая строка объявляет и инициализирует переменную целого типа с идентификатором var. Вторая строка — это вызов оператора форматированного вывода printf, первым аргументом которого является спецификатор строкого типа, второй аргумент содержит конструкцию, возвращающую одну из двух текстовых констант "да" или "нет". Переменная var до символа ? интерпретируется как результат логического выражения. Значение var, отличное от 0, считается логической истиной, 0 – логическая ложь.

Для выбора одного из нескольких значений используется оператор switch. Он описан в пункте 6.8.4.2 стандарта языка программирования. Его форма записи:

```
switch ( выражение )
{
    объявления
    case константа : операторы;
    default: операторы;
}
```

Результатом «выражения», в отличие от оператора if, является значение целого типа. Это делает возможным выбор одного из нескольких значений. В блоке «объявления» возможно объявление и инициализация переменных и функций.

Условие задачи:

Если переменная с идентификатором value равна 2, умножить её на значение переменной value_offset. Если value равна 3, прибавить к ней значение value_offset. Во всех других случаях вычесть из value значение value_offset.

Решение:

```
int value;
...
switch (value)
{
    int value_offset = 4;
    case 2: value *= value_offset; break;
    case 3: value += value_offset; break;
    default: value -= value_offset;
}
```

Пояснение:

Троеточие означает, что часть текста программы пропущена. В области объявления оператора switch объявлена одна переменная целого типа value_offset и инициализирована значение 4.

При выполнении оператора switch производится выбор метки case, соответствующей значению переменной с идентификатором value. Далее выполняются операторы, следующие за этой меткой, пока не будет встречен оператор прерывания break; . После этого управление передается за фигурную скобку, завершающую оператор switch. В случае отсутствия метки case, соответствующей текущему значению выражения, управление передается за оператор switch. В связи с этим настоятельно рекомендуется использование необязательной метки default. Считается, что в случае её отсутствия, программа передает управление за оператор выбора, не выполнив выбора. Такая ситуация часто возникает по причине ошибки значения, переданного в выражение, или ошибки в алгоритме программы.

В примере используется сокращенная форма записи арифметических операций. Выражение

```
value *= value_offset;
```

соответствует

*value = value * value_offset;*

Также интерпретируются остальные операции. Сокращенная форма записи арифметического выражения позволяет реализовать эту операцию за меньшее количество ассемблерных операций.

Условие задачи:

Реализовать реостат с разным сопротивлением на разных участках катушки. Считается, что каждый виток имеет разное значение сопротивления. Всего имеется 5 витков.

Решение:

```
int rheostat_turns;
int resistance;
...
switch ( rheostat_turns )
{
    case 1: resistance += 2;
    case 2: resistance += 3;
    case 3: resistance += 1;
    case 4: resistance += 4;
    case 5: resistance += 2; break;
    default: resistance = 0;
}
```

Пояснение:

Так как в операторе switch в действиях по всем меткам case, кроме case 5, отсутствуют операторы прерывания break, все действия будут выполнены по порядку. Например, при переходе на метку case 2 будут выполнены строки:

case 2: resistance += 3;
case 3: resistance += 1;
case 4: resistance += 4;
case 5: resistance += 2; break;

Будет получено значение *resistance* = 12.

Если будет указано неверное количество витков реостата, будет выполнен переход на метку default и переменной с идентификатором *resistance* будет присвоено значение 0. Чтобы проверить, было ли передано недопустимое значение в качестве выражения оператора switch, необходимо проверить на равенство 0 переменную *resistance* таким образом

if (resistance == 0) было передано неверное количество витков;

3.5. Операторы цикла

Реализация алгоритмов часто требует многократного повторения одних и тех же действий. В языках высокого уровня, к которым относится Си, для этого используются циклы. В стандарте языка Си они именуются итераторами и описаны в разделе 6.8.5 Описание итераторов. Циклов всего три: цикл с предусловием (6.8.5.1. while), цикл с постусловием (6.8.5.2. do), цикл с заданным количеством повторений (6.8.5.3. for). Все циклы содержат в себе выражение, являющееся условием выхода из цикла, и тело цикла. В случае, если тело цикла содержит в себе более одного выражения, необходимо его заключать в кавычки.

Цикл с предусловием имеет следующее описание:

while (выражение) тело цикла;

Для всех циклов используются единые правила обрамления тела цикла фигурными кавычками. Представим два случая, когда тело цикла является одним выражением или несколькими.

while (выражение) выражение1;

или

```
while ( выражение ) {
    выражение 1;
    ...
    выражение n;
}
```

Условие задачи:

Вычерпать весь суп из кастрюли.

Решение:

while (уровень супа в кастрюле > 0) {
зачерпнуть суп;
вылить половник в тарелку;
}

Цикл с постусловием имеет следующее описание

do тело цикла while (выражение);

Условие задачи:

Известно, что дверь закрыта. Открыть дверь вставленным в замок ключом.

Решение:

do {
повернуть ключ в замке;
толкнуть дверь;
}
while (дверь закрыта);

Цикл с заданным количеством итераций имеет следующее описание:

for (выражение 1 ; выражение 2 ; выражение 3) тело цикла;

Выражение 1 используется для задания начального значения счетчика цикла. Выражение 2 является условием выхода из цикла, т.е. пока результат этого выражения не равен 0, тело цикла будет выполняться. Выражение 3 производит увеличение счетчика цикла после каждого выполнения тела цикла.

Условие задачи:

Подпрыгнуть три раза.

Решение:

char count = 0;

for (; count < 3; count++) подпрыгнуть;

Пояснения:

Переменная count является счетчиком цикла. Её объявление и инициализация производится до цикла. Так как в цикле инициализация не требуется, первое выражение содержит пустое выражение, состояние из одного символа, точка с запятой. Второе выражение содержит условие, что счетчик цикла должен быть строго меньше 3. Третье выражение выполняет инкремент счетчика цикла. Далее выполнено развертывание цикла в последовательность действий. Текст описан псевдоязыком с использованием лингвистически недопустимого текста для упрощения, также используются символы /* */ которыми отделяется комментарий от текста программы. При компиляции комментарии компилятором не обрабатываются.

char count = 0; /* Объявление и инициализация счетчика цикла

; выражение 1*/

count < 3; /* выражение 2, count равен 0*/

подпрыгнуть; /* тело цикла*/

count++ /* выражение 3, count равен 0*/

count < 3; /* выражение 2, count равен 1, т. е. он увеличен

в результате предыдущей операции*/

подпрыгнуть; /* тело цикла*/

count++ /* выражение 3, count равен 1*/

count < 3; /* выражение 2, count равен 2, т. е. он увеличен

в результате предыдущей операции*/

подпрыгнуть; /* тело цикла*/
count++ /* выражение 3, count равен 2*/
count < 3; /* выражение 2, count равен 3, т. е. он увеличен
в результате предыдущей операции*/

После того как счетчик цикла стал равен 3, т. е. условию выхода из цикла, выполнение цикла прекращается.

Приведем примеры записи бесконечных циклов. Они часто используются при ожидании событий от аппаратного обеспечения.

Используя цикл с предусловием

while (1) тело цикла;

Используя цикл с постусловием

do тело цикла while (1);

Используя цикл с заданным количеством итераций

for (; 1;) тело цикла;

Как показано выше, циклы взаимозаменяемы. Выбор нужного цикла обычно не вызывает затруднений.

3.6. Функции

При написании небольших учебных программ исходный текст обычно размещается в одной функции main. В соответствии с пунктом 5.1.2.2.1 стандарта языка программирования, выполнение любой программы начинается с первого выражения функции main. Также в этом пункте описаны способы интерпретации аргументов функции main и их значений, передаваемых в программу из ОС. Начиная с пункта 3.1. было показано, что даже самые простые программы не обходятся без вызова библиотечных функций, поставляемых вместе с компилятором. Сама по себе функция — это синтаксически допустимая возможность выделения набора инструкций в отдельную синтаксическую единицу.

К основным причинам использования функций относят:

- выделение повторяющегося исходного текста с возможностью обращения к нему из любого места исходного текста программы;
- выделение специфических частей исходного текста, например, обмен данными с некоторым аппаратным устройством;
- создание с помощью функций логических модулей или библиотек для создания исходного текста, соответствующего разработанному алгоритму;
- повышение читаемости исходного текста программы.

Каждая функция имеет входные и выходные значения, типы которых указываются вместе с идентификатором функции. Входные значения также называются аргументами функции. Сначала указывается тип выходного, результирующего значения, затем

указывается идентификатор функции, далее в скобках перечисляются типы и идентификаторы аргументов, передаваемых в функцию.

Например:

char first_function (float first_argument, int second_argument);

Выходное значение у функции может быть только одно. Это ограничение связано с аппаратной архитектурой большинства современных микропроцессоров, т. к. выходное значение помещается в регистр микропроцессора общего назначения eax, который в логической структуре регистров один. Следует отметить, что в современных микропроцессорах физически может присутствовать сотня регистров eax, однако они недоступны для управления из прикладной программы. Их распределением занимается микропрограмма, зашитая в микропроцессор. В приведенном примере выходное значение имеет тип char. Для возврата значения из функции используется оператор return.

Входные значения функции перед её запуском помещаются в стек, поэтому в функцию можно передать множество аргументов, общий размер которых не превышает размер стека. Входные значения отделяются друг от друга запятой, называемой операцией следования. В приведенном примере входные аргументы — это переменная с идентификатором first_argument типа float и переменная с идентификатором типа second_argument типа int.

Идентификатором функции является first_function.

Функция может быть реализована двумя способами:

1. Объявление и тело функции находится до её

первого использования.

Пример:

```
...
char first_function (float first_argument, int second_argument)
{
    char value;
    ...
    return value;
}

...

int main ()
{
    float float_value;
    int int_value;
    ...
    char result = first_function (float_value, int_value);
    ...
    return 0;
}
```

Из примера видно, названия переменных не должны совпадать с названиями аргументов, передаваемых в функции.

2. Тело функции находится после функции, в которой происходит её вызов. В этом случае необходимо выполнить объявление функции до её вызова. Объявление функции состоит из строки, содержащей тип возвращаемого значения, идентификатора функции, списка входных аргументов, заключенных в скобки, и

точки с запятой.

Например:

```
...
char first_function (float first_argument, int second_argument);
...

int main ()
{
    float float_value;
    int int_value;
    ...
    char result = first_function (float_value, int_value);
    ...
}

...

char first_function (float first_argument, int second_argument)
{
    char value;
    ...
    return value;
}
```

В случае, если выполнено объявление функции, но реализации функции нет, компиляция исходного текста программы возможна, при этом ошибка не будет выдана. Такой прием позволяет проектировать структуру библиотеки или программы, непосредственно написан исходный текст программы, а реализацию отдельных функций разделить между несколькими программистами.

Отсутствие тела функции приведет к ошибке на этапе компоновки. Схема преобразования исходного текста программы в исполняемый файл описана в пункте 3.1.

Функция может не иметь входных аргументов, например:

int main ();

01.c

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

void print_pi ()
{
    printf("%1.20f",M_PI);
}

inline float pi ();

int main ()
{
    printf("pi = ");
    print_pi();
    printf(";\n");
    float pi_value = pi();
    printf("pi = %f;\n", pi_value);
    return 0;
}

float pi ()
{
    return M_PI;
}
```

Рисунок 1 — Демонстрация синтаксиса описания функций

При объявлении функций может использоваться спецификатор inline, описанный в разделе 6.7.4. языка программирования. В этом случае вместо вызова

инструкций функции в строке будут подставлены операции, которые она содержит. Таким образом спецификатор inline позволяет структурировать исходный текст программы, повышать его читаемость, при этом не увеличивая вычислительной сложности программы.

Пример объявления, реализации и использования функций приведен на рисунке 1.

Первая и вторая строка исходного текста программы являются директивами предпроцессора и заменяются на содержимое указанных файлов.

В строке 4 описывается функция с идентификатором print_pi. Функция не принимает никаких значений, потому что не указаны аргументы между открывающей и закрывающей скобкой. Функция ничего не возвращает, поэтому указывается пустой тип void, означающий «ничего». Если функция не возвращает никаких значений, обязательно перед её идентификатором писать void, в соответствии со стандартом, если перед идентификатором функции нет никакого типа, по умолчанию подставляется тип int и компилятор будет требовать указать тип возвращаемого значения.

Реализация функции print_pi состоит из конструкций в 5, 6 и 7 строках. Открывающая фигурная скобка в 5 строке обозначает начало тела функции. Закрывающая фигурная скобка в строке 7 обозначает завершение тела функции. Единственным выражением функции является вызов функции printf стандартной библиотеки ввода/вывода. В первом аргументе оператора printf указывается о необходимости вывести значение с плавающей запятой, при этом целая часть должна состоять из одного символа, а дробная из двадцати. В качестве значения используется константа M_PI, объявленная с помощью оператора предпроцессора #define в файле math.h.

В строке 9 представлено объявление функции pi. Функция не принимает никаких значений, т. к. в скобках отсутствуют аргументы, и возвращает значение типа float. Тело функции не указано за её объявлением и значит для создания исполняемого файла тело функции необходимо реализовать в другом месте. В связи с использованием спецификатора inline, при использовании этой функции её вызов будет заменен инструкциями, из которых она состоит.

В строке 11 объявляется и реализуется третья по счету функция программы, main. Функция не принимает никаких значений и возвращает значение целого типа int. Это главная функция и с неё начинается исполнение программы. Тело функции main ограничено открывающей фигурной скобкой в строке 12 и закрывающей фигурной скобкой в строке 19.

В строке 13 вызывается функция printf модуля стандартного ввода/вывода. Несмотря на то, что в данном вызове не производится формированного вывода, необходимость использования printf обусловлена тем, что требуется произвести вывод без перехода на следующую строку. В свою очередь, оператор puts автоматически дополняет вывод переходом на следующую строку.

В строке 14 выполняется вызов функции print_pi, реализация которой выполнена в строках 4-7.

В строке 15 вызывается функция printf. В данном случае printf мог быть заменен на вызов puts(";"); .

В строке 16 объявляется переменная плавающего типа float с идентификатором pi_value, которая инициализируется результатом выполнения функции pi().

В строке 17 производится вызов функции printf, выполняющей форматированный вывод переменной pi_value в соответствии с шаблоном вывода, переданным первым аргументом `"pi = %f;\n"` .

В строке 18 вызывается оператор return, который помещает в регистр общего назначения eax результат выполнения функции, в данном случае 0, и завершает её работу. В связи с тем, что это главная функция программы, происходит завершение работы программы, а код возврата передается вызывающей программе, т. е. ЭТ.

В строках 21-24 реализована функция pi. В строке 21 содержится её объявление. Оно необходимо для однозначной идентификации реализуемой функции. Тело функции ограничено фигурными скобками, открывающими в 22 строке и закрывающими в 24 строке. Тело функции состоит из вызова оператора return, который возвращает в точку вызова значение константы M_PI, объявленной в файле math.h с помощью директивы предпроцессора #define. Так как при объявлении этой функции в строке 9 использован спецификатор inline при компиляции исходного текста програмы вместо float pi_value = pi(); будет использоваться выражение float pi_value = M_PI;

В строке 25 оставлена пустая строка.

На рисунке 2 представлен результат выполнения программы.

```
[valerii@]$ gcc 01.c -lm && ./a.out
pi = 3.14159265358979311600;
pi = 3.141593;
[valerii@]$
```

Рисунок 2 — Результат выполнения программы, соответствующей исходному тексту программы на рисунке 1

Так как из математической библиотеки math используется только константа, а функции не

используются, при компиляции не обязательно указывать ключ -lm для компоновщика. Предпроцессор заменит строку 2 на содержимое файла math.h, в котором и объявлена константа M_PI. После этого возможна сборка исполняемого файла, т. к. вся необходимая информация для этого будет доступна для компилятора и компоновщика.

В случае возврата значения 1 командой сборки исполняемого файла произойдет запуск собранного файла a.out, находящегося в текущем каталоге.

Вывод первой строки реализован по частям функциями строк 13, 14 и 15. Вывод второй строки, с округленным значением дробной части, производится функцией в строке 17.

Похожие возможности предоставляет механизм, реализованный в ОС под названием разделяемые библиотеки. Он будет рассмотрен в следующих разделах.

3.6. Область видимости

Все переменные хранятся в ОЗУ. Для хранения переменных программ ОС использует два отдельных участка памяти: область стека и область кучи. Стек – участок ОЗУ размером от нескольких килобайт до нескольких мегабайт, выделяемый ОС для программы с целью кратковременного хранения некоторых значений. Куча – все ячейки ОЗУ, не используемые другими программами, включая ОС, и прошивками аппаратного обеспечения. Память на стеке выделяется один раз при создании процесса и ОС не вмешивается в процесс её использования. Размер доступной памяти в области кучи постоянно меняется в зависимости от режимов работы запущенных процессов. ОС контролирует распределение этой памяти, но не вмешивается в то, как выделенная память используется программой для своих нужд, поэтому память в области кучи также называют динамической памятью.

В примерах, приводимых ранее, все переменные объявляются и инициализируются на стеке. Например:

int i;

Для создания переменной в области кучи используется одна из специальных функций. Память, выделенную в области кучи, необходимо очищать от предыдущих значений и освобождать до завершения выполнения программы. Более подробно работа с памятью в области кучи будет рассмотрена в соответствующем разделе.

Программа gcc может выполнять дезасамблерирование исходного текста программы на языках программирования в ассемблерный текст

нескольких нотаций. Для этого используется ключ -S, например:

gcc 1.c -S -masm=intel

В результате дезасемблирования программы, содержащей выражения на языке Си, которые объявляют две переменные и инициализируют их некоторыми значениями

```
int i = 0;
int j = 2;
```

был получен текст программы на языке ассемблер в синтаксисе masm

```
mov DWORD PTR [rbp-12], 0
mov DWORD PTR [rbp-8], 2
```

Обычно одной строчке на языке Си соответствует множество команд на языке ассемблер, однако в приведенном примере соответствие однозначное. Не вдаваясь в подробное описание работы со стеком с помощью ассемблерных инструкций отметим, для хранения данных на стеке идентификаторы переменных не используются. Переменная хранится как смещение от максимального значения в регистре микропроцессора rbp. Экскурс в ассемблерные инструкции сделан, чтобы аргументировано показать, что идентификатор переменной нужен только для удобства при работе с исходными текстами программ. Следствием из этого вывода является возможность использовать один и тот же идентификатор для нескольких переменных. Этот прием настоятельно не рекомендуется и категорически

запрещен для начинающих программистов!

Наиболее часто один и тот же идентификатор переменных используется для счетчиков циклов, указателей на временные ресурсы. Этот подход оправдан только в том случае, если программа пишется на основе хорошо написанного технического задания, редактирование которого не предполагается. В противном случае могут возникать сложные для поиска и исправления ошибки.

Пример использования одного идентификатора для различных переменных

```
int i;
if ( условие ) {
    int i;
    ...
}
```

02.c

```
1  #include <stdio.h>
2
3  int main () {
4      int i=0;
5      printf ("%d\n",i);
6      {
7          int i=1;
8          printf ("%d\n",i);
9          {
10             int i=2;
11             printf ("%d\n",i);
12         }
13         printf ("%d\n",i);
14     }
15     printf ("%d\n",i);
16     return 0;
17 }
18
```

Рисунок 1 — Исходный текст программы, демонстрирующей область видимости переменных

После прохождения открывающей фигурной скобки, появляется возможность использовать такой же идентификатор переменной, однако значение, объявленное до открывающей фигурной скобки, становиться недоступным.

Если текст программы на рисунке 1 непонятен, необходимо перечитать построчные описания исходных текстов программ пунктов 3.1-3.5. Далее будут даны пояснения, касающиеся только области видимости.

Объявленная и инициализированная переменная целого типа с идентификатором i после выполнения выражения в строке 4 будет равна 0. Оператор printf в пятой строке выведет значение 0. В седьмой строке объявлена и инициализирована переменная с идентификатором i, однако эта переменная расположена в ячейках памяти, отличных от переменной, объявленной в строке 4, т.к. между этими переменными присутствует открывающая фигурная скобка.

Открывающая фигурная скобка условно может считаться операцией со стеком. В языке Си переменная может быть объявлена в любом месте, однако фактически операции создания переменных на стеке производятся или в начале программы или группируются в месте, примерно соответствующем открывающей фигурной скобке. Соответственно, уничтожаются они в конце программы или в области закрывающей фигурной скобки. Здесь под уничтожением подразумевается изменение указателя области стека, который отделяет используемую память стека от неиспользуемой. Гарантированного затирая ячеек памяти не происходит, а зависит от аппаратного обеспечения, версии ОС, используемой версии компилятора, библиотек, используемых для создания исполняемого файла.

```
[valerii@]$ gcc 02.c -o 02 && ./02
0
1
2
1
0
[valerii@]$
```

Рисунок 2 — Создание исполняемого файла на основе исходного текста программы рисунка 1, его исполнение

На рисунке 2 представлено содержимое потока вывода программы, полученной из исходного текста рисунка 1. В пункте 3.3 подробно разобрана команда, с помощью которой получается исполняемый файл.

Первая выведенная цифра 0 выведена функцией printf строки 5. Цифра 1 выведена функцией строки 8. Цифра 2 выведена функцией строки 11. В четвертой строке вывода выведена цифра 1, соответствующая 13 строке исходного текста программы. Последняя строка вывода содержит 0 и соответствует 15 строке исходного текста программы.

Для управления областью видимости используются ключевые слова static, extern. Данные спецификаторы могут использоваться как для переменных и констант, так и для функций. В пункте 6.9.1 стандарта языка программирования описано внешнее определение для функций, которое используется реже, чем внешние переменные или константы.

Рассмотрим пример исходного текста программы, суммирующей минимальные значения или любое значение, если входные аргументы равны. Для учебных целей он написан в несколько запуанной манере,

свидетельствующей о плохом структурировании решения поставленной задачи. Признаками плохой структуры в данном случае является вызов функции из одного файла, реализованную в другом файле, которая, в свою очередь, вызывает функцию, расположенную в первом файле, с исходным текстом главной функции.

03_1.c | 03_2.c | 03_2.h

```
1  #include <stdio.h>
2  #include "03_2.h"
3
4  int min (int a, int b)
5  {
6      return a<b?a:b;
7  }
8
9  int main ()
10 {
11     int a = 3;
12     int b = 4;
13     int variable = 0;
14     variable = calc (a, b);
15     printf("%d = calc (%d, %d);\n", variable, a, b);
16     printf("%d = calc (%d, %d);\n", calc (a, b), a, b);
17     return 1;
18 }
19
```

Рисунок 3 — Текст программы файла 03_1.c

В строке 1 используется директива предпроцессора, которая будет заменена предпроцессором на содержимое файла stdio.h, расположенного в каталоге include поставляемого вместе с компилятором. На это указывают скобки в виде знаков < и > .

В строке 2 директива предпроцессора указывает на необходимость включения в текст программы содержимого заголовочного файла 03_2.h, который будет описан далее. Файл 03_2.h должен быть расположен в том же каталоге, что и файл 03_1.c. На это указывают скобки в виде двойных кавычек.

В строке 4 объявление и реализация функции с идентификатором min, принимающая два аргумента целочисленного типа с идентификаторами a и b и возвращающая целочисленное значение. Функция выполняет поиск минимального значения из двух переданных. Тело функции ограничено фигурными скобками. Открывающая скобка стоит в строке 5, закрывающая скобка в строке 7.

В строке 6 находится единственная строка функции min, являющаяся вызовом оператора return, которому в качестве аргумента передается оператор выбора, состоящий из символа « ? » и символа « : ». Оператор выбора выполняет выражение a < b. Если условие верное, т. е. a — наибольшее значение, значит оно будет возвращено оператору return, во всех других случаях оператору return будет передано значение b. Оператор return завершит выполнение функции min и вернет значение в точку вызова.

В строке 9 объявление главной функции программы. Тело функции ограничено фигурными скобками. Функция начинается в строке 10 и завершается в строке 18.

В строке 11 объявлена целочисленная переменная типа int с идентификатором a, инициализированная значением 3.

В строке 12 объявлена целочисленная переменная типа int с идентификатором b, инициализированная значением 4.

В строке 13 объявлена целочисленная переменная типа int с идентификатором variable, инициализированная значением 0.

В строке 14 вызывается функция calc, объявление которой выполнено в файле 03_2.h. В функцию в качестве аргументов передаются значения переменных a и b, а результат выполнения функции присваивается

переменной с идентификатором variable.

В строке 15 вызывается функция стандартной библиотеки ввода/вывода printf. Первым аргументом в неё передается шаблон вывода, содержащий три спецификатора типа %d, которые будут заменены значениями, переданными 2, 3, 4 аргументами.

Строка 16 соответствует вызову строки 14 и 15, однако запись сделана в сокращенном виде в одну строку. Первым аргументом в функцию printf передается шаблон вывода, идентичный шаблону вывода строки 15. Второй аргумент состоит из вызова функции cal объявленной в заголовочном файле 03_2.h. В свою очередь, в функцию calc передаются аргументы, значения переменных a и b. Аргументы 3 и 4 функции printf соответствуют аргументам в строке 15.

В строке 17 вызывается оператор return, в который передается значение 1. Оператор завершит выполнение функции main. Функция main является главной функцией программы, значит программа будет завершена, а значение 1 будет возвращено в вызвавшую её программу, а именно в ЭТ.

Строка 19 соответствует пустой строке в конце файла.

В этом файле функция min объявлена и реализована, однако нигде не использована. А функция calc наоборот, использована, но нигде не объявлена.

03_1.c | 03_2.c | 03_2.h

```
1  #include <stdio.h>
2
3  int calc (int a, int b);
4  extern int min (int a, int b);
5
```

Рисунок 4 — Текст заголовочного файла 03_2.h

Файл 03_2.h является заголовочным файлом. В строке 1 использована директива предпроцессора, которая будет заменена содержимым файла stdio.h, расположенного в каталоге include поставляемого вместе с компилятором.

В строке 3 объявляется функция с идентификатором calc, которая принимает два целочисленных аргумента типа int с идентификаторами a и b, и возвращает целочисленное значение. Однако тело функции в этом файле также отсутствует.

В строке 4 с помощью спецификатора extern объявляется внешняя функция с идентификатором min, которая была объявлена в файле 03_1.c, где и представлено её описание и реализация.

Заголовочный файл и файл с исходным текстом программы отрабатываются как единое целое, в случае, если их имена до расширения имени файла, т. е. до последней точки, совпадают.

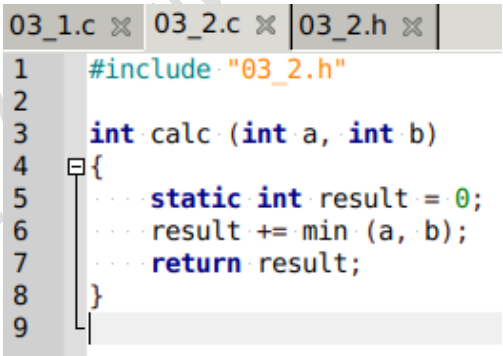


Рисунок 5 — Текст программы файла 03_2.c

В файле 03_2.c в первой строке указана директива предпроцессора, которая будет заменена содержимым файла 03_2.h . Благодаря этому в файле 03_2.c доступна функция min, а функция calc становится доступна при подключении в файле заголовочного файла 03_2.h так,

как это сделано в 03_1.c.

В строке 3 объявление функции calc, совпадающее с объявлением этой функции в файле 03_2.h. Далее следует тело функции calc, которое ограничено открывающей фигурной скобкой в строке 4 и закрывающей фигурной скобкой в строке 8.

В строке 5 объявлена целочисленная переменная с идентификатором result и спецификатором static, которая инициализирована значением 0. Использование спецификатора static позволяет создать переменную, ячейки памяти которой не будут освобождаться каждый раз после её завершения.

В строке 6 использована сокращенная форма записи += , которая означает прибавление некоторого значения к переменной слева от этой операции. Справа от этого знака вызывается функция min, которая реализована в файле 03_1.c . В функцию min передаются аргументы a и b, значения которых получено функцией calc из вызывающей её функции.

В строке 7 вызывается оператор return, возвращающий вычисленное значение result в точку вызова функции min.

Следует отметить, в файле 03_2.c не используются элементы стандартной библиотеки ввода/вывода, однако в заголовочном файле 03_2.h указана директива подключения этой библиотеки. Такое объявление называется рудиментным. Обычно такое встречается в тексте программы, если текст изначально был плохо структурирован или переписывался. При решении практических задач практически всегда остается рудиментный текст.

Команда получения исполняемого файла, его запуск и результат исполнения представлен на рисунке 6. Для компиляции программы, состоящей их нескольких

файлов, необходимо указывать их все в определенном порядке. Сначала указываются файлы с конечными функциями, затем те, которые их используют, и в конце файл с функцией main. В остальном строка соответствует тем, которые использовались в предыдущих примерах.

```
[valerii@]$ gcc 03_2.c 03_1.c -o 03 && ./03
3 = calc (3, 4);
6 = calc (3, 4);
[valerii@]$
```

Рисунок 6 — Создание исполняемого файла из 03_1.c, 03_2.h, 03_2.c и результат его исполнения

Первая строка вывода сформирована вызовом функции printf в строке 15 файла 03_1.c , которая произошла после первого вызова функции calc. Таким образом, результатом первого вызова функции calc с аргументами a=3 и b=4 является 3. Проверка. Меньшим из a и b является a=3. Начальным значением result в файле 03_2.c является 0. 3+0=3.

Вторая строка вывода сформирована строкой 16 файла 03_1.c , которая выводит значение после второго вызова функции calc. Второй вызов функции calc произведен с теми же аргументами что и первый, однако получен результат 6. Это связано с сохранением предыдущего результата работы функции calc в переменной result, которая объявлена со спецификатором static и доступна только внутри функции calc. Проверка. Меньшим из a и b является a=3. Текущим значением result в файле 03_2.c является 3. 3+3=6.

Изменение результирующего значения неочевидно и требует поиска функции, которая приводит к таким результатам и просмотру исходного текста этой функции. Избежать подобных проблем позволяет предварительное

проектирование программы и подробное комментирование назначения реализуемых функций.

3.8. Операции прерывания и безусловного перехода

В пункте 6.8.6 стандарта языка программирования описаны операторы безусловного перехода. К ним относятся: **goto**, **continue**, **break**, **return**. Приведем пример исходного текста программы, демонстрирующей использование операторов безусловного перехода.

01.c

```
1  #include <stdio.h>
2
3  int main () {
4      int i, summ, max_value;
5      i = summ = max_value = 0;
6      scanf("%d", &max_value);
7      for ( ; i < max_value; i++) {
8          if (i % 2) continue;
9          summ += i;
10     }
11     printf("%d\n", summ);
12
13     for (i = 0; i < max_value; i++) {
14         if (i % 2) goto lab;
15         summ += i;
16         lab:
17         if (summ > 1000) break;
18     }
19     printf("%d\n", summ);
20     return 0;
21 }
22
```

Рисунок 1 — Исходный текст программы, демонстрирующей операторы безусловного перехода

В строке 1 используется директива предпроцессора, предписывающая предпроцессору заменить её на содержимое файла stdio.h, находящегося в каталоге include, поставляемом вместе с компилятором.

Строка 2 является отступом для визуального разграничения команды предпроцессора от команд для

компилятора.

Строка 3 — это начало главной функции main, с которой начинается выполнение всех программ. Функция не принимает никаких аргументов, т. к. в скобках ничего не написано, и возвращает значение типа int в вызывающую её программу. Тело функции main ограничено открывающей фигурной скобкой в строке 3 и закрывающей фигурной скобки в строке 21.

Строка 4 содержит объявление трех переменных целочисленного типа int с идентификаторами i, summ, max_value. Идентификаторы разъедены символом запятая, которая называется операцией следования. Операция следования также используется при перечислении аргументов функции.

В строке 5 производится множественное присвоение. Такие строки читаются справа налево. В данном случае значение 0 будет присвоено переменной с идентификатором max_value, затем значение max_value будет присвоено переменной summ. Далее значение переменной summ будет присвоено значению переменной i. Компилятор при обработке множественного присваивания значение справа помещается в переменные непосредственно из регистра микропроцессора, что позволяет сэкономить как минимум одну ассемблерную инструкцию.

В строке 6 вызывается функция считывания из стандартного потока ввода scanf. Её первым аргументом является шаблон ввода, соответствующий целочисленному значению. Второй аргумент состоит из идентификатора переменной max_value и унарной операции взятия адреса &. В результате выполнения выражения во втором аргументе функции scanf, в эту функцию будет передан указатель на переменную max_value, по которому scanf запишет значение,

считанное из стандартного потока ввода, предварительно отформатировав его в соответствии с шаблонов ввода из первого аргумента.

В строке 7 написан цикл for, цикл с заданным количеством итераций. Первое выражение цикла пропущено. В начале обучения программированию так делать не нужно. Легче лишний раз сбросить счетчик цикла с помощью первого выражения, чем потратить значительное время на поиск ошибки. Второе выражение указывает на то, что цикл будет выполняться, пока счетчик цикла — переменная с идентификатором i меньше переменной max_value. Третье выражение устанавливает скорость изменения счетчика цикла с помощью операции инкремента i++, который соответствует увеличение на единицы. Отличие инкремента в том, что он выполняется как одна инструкция микропроцессора, а увеличение значения переменной на единицу как три инструкции. Тело цикла for ограничено открывающей фигурной скобкой в строке 7 и закрывающей фигурной скобкой в строке 10.

В строке 8 используется оператор условного перехода if. Для его исполнения сначала вычисляется выражение, состоящея из бинарной операции получения остатка от деления i на 2. Если остаток от деления не равен 0, выполняется выражение после оператора if. Так как отсутствует фигурная скобка, будет выполнено только одно выражение, следующее за оператором if(). Этим выражением является оператор безусловного перехода continue, который прерывает выполнение текущей итерации цикла, т. е. осуществляет действие, идентичное переходу на закрывающую скобку текущей итерации цикла. После этого выполняется третье выражение i++ оператора for в строке 7, затем второе выражение в строке 7 как проверка условия завершения

цикла. В случае, если i < max_value, происходит переход в тело цикла в строку 8 и т.д.

В строке 9 используется сокращенная запись арифметического выражения, полная версия которого выглядит так: summ = summ + i; , т. е. производится увеличение значения переменной summ на величину i.

В строке 11 вызывается оператор форматированного вывода printf. Первый аргумент состоит из шаблона целочисленного вывода и служебной последовательности \n, соответствующей переходу на новую строку. В результате выполнения этой строки в стандартный поток вывода будет выведено значение переменной summ и выполнен переход на новую строку.

В строке 13 описан оператор с заданным количеством итераций for. В отличие от предыдущего использования в строке 7, в этом случае с помощью первого выражения i = 0 задается начальное значение счетчика цикла. Второе и третье выражение соответствует циклу for строки 7.

В строке 14 использован оператор условного перехода if как и в строке 8. Выражением после оператора условного перехода является оператор безусловного перехода goto, который предписывает начать выполнять выражения после метки lab в строке 16. Если остаток от деления i на 2 равен 0, goto не будет выполнено и будет выполнена строка 15.

Строка 15 соответствует строке 9.

В строке 17 оператор условного перехода if предписывает выполнить оператор break в случае, если значение переменной summ больше 1000. Оператор break прерывает выполнение цикла или оператора множественного выбора switch.

В строке 19 вызывается оператор printf, соответствующий вызову в строке 11.

В строке 20 использован оператор return, возвращающий значение 0 в вызывающую функцию, т. к. оператор использован в функции main, возврат значения произойдет в вызывающую программу, ЭТ.

```
[valerii@]$ gcc 01.c -o 01 && ./01
11
30
60
[valerii@]$ gcc 01.c -o 01 && ./01
100
2450
2450
[valerii@]$
```

Рисунок 2 — Создание исполняемого файла на основе исходного текста программы рисунка 1, его исполнение

На рисунке 2 представлен результат двухкратного вызова программы, полученной из исходного текста программы на рисунке 1. Первая команда вызывает программу, выполняющую создание исполняемого файла из исходного текста программы и передает ей в качестве аргументов имя файла с исходным текстом, ключ -o, указывающий имя создаваемого исполняемого файла, в данном случае 01. Далее использована логическая операции И, второй аргумент которой является вызовом созданного исполняемого файла из текущего каталога.

После запуска программы её исполнение остановится в строке 6, пока не будет введено значение и не нажата клавиша Enter. Если ввести значение 11, то будет выполнен расчет сумм всех четных чисел до 11, т. е. 0+2+4+6+8+10 = 30. За суммирование отвечает текст программы 7-10 строк.

Вторая строка в терминале будет выведена оператором printf строки 11. Далее будет выполнен

второй цикл, в котором оператор continue заменен оператором goto и меткой lab: , а также добавлено условие прерывания цикла for в строке 17, предписывающее прекратить выполнять цикл, если при исполнении строки 17 значение переменной summ будет более 1000.

Третья строка вывода сформирована оператором printf в строке 19.

Таким образом при первом запуске программы посчитана сумма всех четных чисел до введенного числа 11, которая равна 30. Второй цикл также посчитал сумму всех четных чисел до 11, которая равна 30, прибавляя постоянно значения к переменной summ. Таким образом программа рассчитала удвоенную сумму всех четных чисел до 11.

Второй запуск не требовал повторного создания исполняемого файла, т. к. изменения в него не вносились. При повторном запуске программы входным значением стало 100. Сумма всех четных чисел до 100 равна 0+2+4+...+98 = 2450. Удвоить рассчитанную сумму не получилось, т. к. она больше 1000, а значит оператор break прекратил выполнение второго цикла при выполнении первой итерации цикла.

Следует обратить внимание на тот факт, что операция получения остатка от деления от 0, в отличие от деления на 0, не вызывает арифметического исключения и программа продолжает работать.

В стандарте языка программирования также описаны и другие операторы прерывания выполнения программы. Например, **abort()** (пункт 7.22.4.1) и **exit()** (пункт 7.22.4.4) библиотеки stdlib, поставляемой вместе с компилятором. А также ряд функций, схожих с ними по действию: **_Exit()** (пункт 7.22.4.5), **quick_exit()** (пункт 7.22.4.7).

Функции семейства **at_exit (void (*func) (void));** (пункт 7.22.4.2) позволяют задать функцию, которая будет вызываться перед нормальным завершением программы. Её можно использовать для освобождения всей выделенной в процессе работы программы памяти или освобождения других ресурсов, т.е. для сборки мусора. Под мусором здесь подразумевается использованные ресурсы, которые больше не нужны для выполнения инструкций программы.

В разделе 7.22.4 Коммуникация с окружением, есть описание других функций, схожих с приведенными ранее.

3.9. Стиль программирования

Стиль программирования - набор правил оформления исходного текста программы, предпочитаемые алгоритмы и паттерны программирования. Он уникален для каждого программиста, также по нему можно определить, какой язык является родным для программиста, его осведомленность в различных областях теории программирования и некоторую другую информацию.

В данном разделе под стилем программирования подразумевается исключительно правило оформления исходного текста программы. На сегодняшний день наиболее популярными являются две нотации: **UNIX нотация** и **венгерская нотация**. Первая из них поддерживалась сообществом UNIX на протяжении десятилетий. Вторая поддержана MicrosoftTM и разработчиками, программирующими на языке C++. Следует отметить, язык C++ не является надмножеством языка Си. На сегодняшний день, в целях недопущения ошибок лучше считать, что C++ также похож на Си, как и PHP.

К базовым правилам стиля программирования следует отнести следующее:

- исходный текст программы пишется для компилятора, а оформляется для человека;
- безобразно, зато единообразно;
- чем более насыщенный текст программы, тем проще сравнивать объем и сложность, однако сложнее читать и изменять.

Чтобы сформировать собственный стиль программирования, необходимо взять за основу стиль, предложенный авторами языка. Он исчерпывающе представлен в книге «Язык программирования Си»

Брайена Кернигана и Денниса Ритчи. В кругах разработчиков он называется K&R. Только в этом случае можно предпринять попытку понять, какие проблема решает Си, в чем его красота и почему миллионы строк текста написаны именно на нем.

Одним из требований при написании исходного текста программы является соблюдение принятого в данной группе разработчиков стиля программирования. Часто он изложен в соответствующем руководстве или рекомендуется некоторый набор исходных текстов для образца. Использование в группе разработчиков одного стиля позволяет упростить чтение исходного текста программ, когда структура файла схватывается налету и требует внимательного прочтения всего текста.

Приведем рекомендуемый набор правил для оформления исходного текста программ.

1. Длина строки не должна превышать 100 строк.

2. Для задания идентификаторов типов, констант, переменных, указателей, функций должны использоваться разные правила. В идентификаторах стоит избегать аббревиатур, двояко понимаемых сокращений и слишком коротких названий.

Например:

#typedef int digit; // Переопределение типа. Все типы

// пишутся строчными буквами.

#typedef MAX_BUFF_SIZE 164 // Идентификаторы констант

// пишутся заглавными буквами

*int *ptr_counter; // Указатели выделяются с помощью*
// последовательности символов «_ptr».
struct telnomber {
char country_code; // Определение переменной в UNIX нотации.
// Разделителем слов в идентификаторе
// является символ «_».
digit nomber;
};
*void set_telnomber (struct telnomber * tel_nom); // Название функции*
// содержит слово действие set,
// разделителем слов является «_».

3. При объявлении экземпляров объектов использовать похожие идентификаторы.

Например:

*void set_telnomber (struct telnomber * tel_nom); // Название аргумента*
// tel_nom, созвучно своему типу struct telnomber
// и функции set_telnomber().

4. Каждая переменная объявляется в отдельной строке.

Например:

char country_code;
int nomber;

5. Всегда использовать отступы в места, где по синтаксису языка используются или подразумеваются фигурные скобки. По всему тексту программы используются отступы одинаковой кратности, т. е. все отступы кратны 4 пробелам. Количество пробелом можно выбирать самостоятельно. Отступы в 2 пробела плохо различимы при исходных текстах более 100 строк и использовании шрифтов, отображающих пробелы уже остальных символов. Отступы, кратные более 6 пробелам использовать слишком расточительно, из-за ограничений на ширину строки. Текстовый редактор среды Geany отводит на строку 72 символа, проводя перед 73-м символом непечатаемую вертикальную серую черту. Для максимальной переносимости используются пробелы. Горизонтальная табуляция может по разному отображаться в каждом экземпляре редактора, таким образом исходный текст с табуляцией будет удобно читать только в том редакторе на той ЭВМ, в котором он написан. Однако идеологически, горизонтальная табуляция является более правильным решением, так как она является специальным символом, специально изобретенным для этой цели.

```
If (a < d)
   assert (a);
else {
   if (a == d) {
      assert ("стороны многоугольника равны");
   }
   else {
      assert (d);
   }
}
```

6. Пробелы необходимо использовать не только для соблюдения синтаксических правил, но и для улучшения читаемости исходного текста программы. В некоторых случаях они избавляют от синтаксических ошибок (j++ +i). В правиле 1 комментарии написаны с выравниванием пробелами для повышения читаемости. Рекомендуется использовать пробел между типом и символами «*» или «&», но не использовать между этими операциями и идентификаторами переменных (int *digit; digit = &value).

Пример неправильного форматирования:

```
If(a<d)assert(a);else{if(a==d){assert ("стороны многоугольника равны");}else{assert(d);}}
```

Пример:

```
#define MAX_TEMP      94   // Максимальная температура
#define MIN_TEMP      11   // Минимальная температура
#define START_TEMP    34   // Начальная температура
#define STOP_TEMP     80   // Конечная температура
```

7. Использовать фигурные скобки, когда тело функции пустое или отсутствуют выражения.

Например:

```
void send_dev_null (char value) {};
if (mass > 0) {}
else assert(mass);
```

8. Использовать круглые скобки для группировки выражений по смыслу и для упрощения понимания последовательности операций, даже если этого не требует приоритет операций.

Например:

```
if (((a > 0) && (b > 0)) || (c < 10)) ...
```

9. Всегда пишите оператор switch, используя одинаковый шаблон форматирования.

Например:

```
switch (value) {
   case 1:
      function_1 ();
      break;
   case 2:
   case 3:
      function_2 ();
      break;
   default:
      function_1 ();
}
```

10. Объем текста функции не более одного экрана (не более 50 строк). Если в функции более 50 строк, обычно означает, что это решение задачи плохо структурировано.

11. В случае переноса аргументов вызываемой функции или разрыва строки необходимо делать значительный отступ пробелами. Такие фрагменты текста программы становятся зрительными «якорями», за которые цепляется взгляд и значительно упрощает

ориентирование в тексте программы.

Например:

```
int result = function_1 (arg_1,
                         arg_2,
                         arg_3,
                         arg_4);
```

12. Все объявления функций и глобальных объектов необходимо производить в заголовочных файлах. Локальные переменные необходимо объявлять непосредственно перед её первым использованием.

13. Комментарии не следует писать в конце строки так, чтобы они были не видны.

14. В начале файла нужен многострочный комментарий, описывающий что делает текст программы этого файла, его лицензионные ограничения. Если необходимо, указывается способ использования в составе программного комплекса или особенности компиляции.

15. Для визуального разделения блоков исходного текста желательно использовать пустые строки.

Для повышения переносимости текста программ необходимо также использовать правила MISRA C, которые не являются обязательными к исполнению, но следованием им позволяет решить ряд проблем.

4. Язык программирования Си. Составные типы данных

Человеческий мозг, по разным оценкам, может оперировать одновременно 5 – 8 значениями. С другой стороны, большинство задач для своего решения требует значительно большее количество значений. Одним из способов преодоления этого противоречия является использование наборов данных, объединяемых по разному признаку.

Однотипные данные, в соответствии с определением типов, часто обрабатываются одинаковыми последовательностями команд. Для такой обработки используются составные типы — массивы, и имитируемые с помощью адресной арифметики многомерные массивы.

Отдельно выделяется обработка текста. Обработка текста является одним из важнейших направлений для программирования. Текст состоит из однотипных значений, поэтому для его обработки также используются массивы.

Значения разных типов могут описывать один объект, одно событие или явление. В таком случае часто их обработка производится в одной функции или одном модуле. В этом случае используются структуры.

В некоторых случаях значение одного типа необходимо интерпретировать как значение другого типа. Для разной интерпретации одной и той же области ОЗУ, а также для экономии ОЗУ используется тип объединение.

Для повышения читаемости исходного текста программы некоторые константы можно заменить на сокращения или аббревиатуры. Для этого используется тип перечисления.

В языке программирования существует тип битовые поля, который используя ряд операций позволяет изменять значения отдельного бита, что позволяет экономить память ОЗУ, однако требует больше микропроцессорных операций.

4.1. Составные типы данных: массивы

4.1.1. Объявление и инициализация массивов

Массив – набор объектов одного типа, расположенных в ОЗУ последовательно друг за другом без промежутков.

Формат объявления массивов приведен в пункте 6.7.6.2 стандарта языка программирования. Приведем объявление одномерного массива из 5 элементов, каждый из которых имеет тип целого числа int.

int array[5];

В начале выражения, объявляющего массив, указывается тип элементов массива (int), далее следует идентификатор переменной (array). Затем в квадратных скобках указывается количество элементов массива – 5.

Объявленный массив состоит из пяти элементов. Для обращения к каждому элементу необходимо использовать идентификатор массива и индекс, указывающий на смещение от начала массива. Таким образом, после объявления массива будут доступны следующие элементы:

array[0] , array[1] , array[2] , array[3] , array[4]

Еще раз обратим внимание, при инициализации

массива указывается количество элементов массива. При обращении к элементам массива используется смещение от начала массива. Подробно ознакомится с таким подходом можно на примере абстракции «машина Поста».

Инициализация значений массива константными значениями описана в пункте 6.7.9. стандарта языка программирования. Следует учесть, в этом разделе описаны все возможные случаи для разных составных типов, поэтому как не парадоксально, без знания языка программирования сложно понять, что там написано. В данном разделе приведено максимально простое описание. Для задания начального значения каждому элементу массива используются различные приемы, однако они доступны только в момент объявления массива.

int array[5] = {1, 2, 3, 4, 5};

Выполненная таким образом инициализация приведет к заданию следующих значений.

array[0] = 1, array[1] = 2, array[2] = 3, array[3] = 4, array[4] = 5.

Если при инициализации задается каждое значение, размер массива можно не указывать. Компилятор сам посчитает количество элементов массива.

int array[] = {1, 2, 3, 4, 5};

В приведенном примере массив с идентификатором array будет состоять из 5 символов.

С помощью фигурных скобок может производится

инициализация нескольких элементов массива. В этом случае, остальным значениям будет присвоено значение 0. Однако не стоит забывать указывать правильное количество элементов массива.

```
int array[5] = {1, 2};
```

Приведенная инициализация приведет к заданию следующих значений:

array[0] = 1, array[1] = 2, array[2] = 0, array[3] = 0, array[4] = 0.

Обнуление значений массива при инициализации является следствием предыдущего примера:

```
int array[5] = {0};
```

Данная инициализация задает значение элемента с индексом 0 равное 0, а все последующие значения также равны 0. В результате все элементы массива будут равны 0.

array[0] = 0, array[1] = 0, array[2] = 0, array[3] = 0, array[4] = 0.

Для задания значений элементам массива используется цикл.

```
int array[10] = {0};
int count = 0;
for ( ; count < 10; count++)
      array[count] = count;
```

Начиная со стандарта ISO/IEC 9899:1999 в языке Си появилась возможность объявлять массивы, количество элементов в которых задается в режиме исполнения, т. е. в момент работы программы.

int array[n];

Для корректной работы приведенного примера необходимо чтобы значение n было задано до строки, инициализирующей массив с её использованием.

Приведенные выше примеры создают массив на стеке. Область стека ограничена и не позволяет использовать все пространство ОЗУ. Для массивов с количеством элементов более 50 рекомендуется использовать память в области кучи. При использовании более 500 элементов, настоятельно рекомендуется использовать память в области кучи, т. е. динамически выделяемой памяти.

Выделение и освобождение памяти в области кучи считаются медленными операциями, поэтому не рекомендуется использовать динамическую память для переменных, состоящих из одного значения. При использовании массива операция выделения применяется один раз и делает доступными множество переменных, поэтому использование медленной операции допустимо.

Выделение памяти для размещения массива будет представлено в следующем разделе.

4.1.2. Обращение к элементам массива

При работе с массивами возможно получение доступа к его элементам как последовательно, так и выборочно к произвольному элементу. При этом последовательный доступ является многократным выборочным доступом с последовательно увеличивающимся счетчикам смещения.

Основными операциями при работе с массивами являются: инициализация значений и изменение произвольного элемента массива.

Рассмотрим исходный текст программы, инициализирующий элементы массива, изменяющий два элемента массива и выводящий результат в стандартный поток вывода после каждого набора операций с массивом. Исходный текст программы состоит из трех файлов: 01.c — текст программы, array_output.h — заголовочный файл функции вывода в стандартный поток вывода, array_output.c — текст программы функции вывода в стандартный поток вывода.

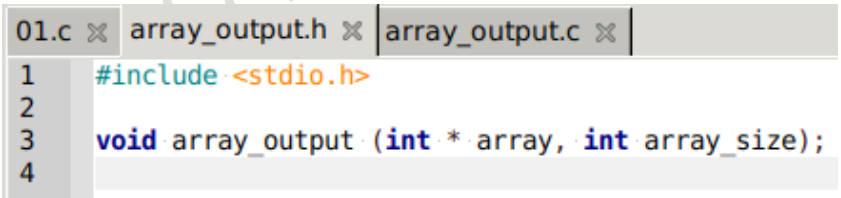


Рисунок 1 — Исходный текст файла заголовка array_output.h

Первая строка на рисунке 1 является директивой предпроцессора. При её обработке она заменяется на содержимое файла stdio.h, расположенного в каталоге include поставляемого вместе с компилятором.

Третья строка объявляет функцию с

идентификатором array_output, принимающую указатель на объект типа int с идентификатором array и переменную целочисленного типа int с идентификатором array_size. Функция не возвращает никаких значений. Из названий аргументов функции следует, что первый аргумент является указателем на массив, а второй сообщает, сколько элементов содержится в массиве. Более подробное объяснение того почему указатель на переменную целочисленного типа может быть интерпретирован как указатель на массив, будет в следующих разделах.

Перейдем к исходному тексту программы на рисунке 2.

Строка 1. В ней использована директива предпроцессора, которая на первом этапе компиляции будет замена на содержимое заголовочного файла array_output.h, представленного на рисунке 1. Имя файла указано в двойных кавычках, поэтому заголовочный файл должен быть расположен в том же каталоге, что и файл array_output.c. С помощью этой строки подключается стандартная библиотека ввода-вывода, которая необходима для использования функций puts и printf.

В строке 3 указан идентификатор функции, принимаемые аргументы и возвращаемое значение с целью определения однозначного соответствия реализуемой функции объявленному ранее прототипу.

Строка 4 содержит один символ открывающую фигурную скобку с которого начинается тело функции array_output.

В строке 5 объявляется целочисленная переменная с идентификатором count, которая инициализируется значением 0. Всегда, независимо от стажа программирования, необходимо инициализировать все объявляемые переменные.

01.c | array_output.h | array_output.c

```
1   #include "array_output.h"
2
3   void array_output (int * array, int array_size)
4   {
5       int count = 0;
6       for (count = 0; count < array_size; count++) {
7           printf("array[%d] = %d", count, array[count]);
8           if (count < array_size-1) printf(", ");
9       }
10      puts(".");
11  }
12
```

Рисунок 2 — Исходный текст функции вывода массива на экран array_output.c

В строке 6 объявлен цикл for, счетчиком цикла которого является переменная с идентификатором count. Начальным значением счетчика является 0. Несмотря на то, что счетчик цикла был инициирован строкой выше, всегда, за очень редким исключением, необходимо инициировать счетчик цикла заново. Причина в том, что в процессе редактирования исходного текста программы в течение длительного периода времени, текст программы может измениться произвольным образом и нет гарантии, что между строкой инициализации и циклом не появится еще несколько циклов.

Условие завершения выполнения цикла count < array_count. Шаг приращения инкремент, т. е. плюс 1. Тело цикла for ограничено открывающей фигурной скобкой в строке 6 и закрывающей фигурной скобкой в строке 9.

Тело цикла for состоит из двух выражений, поэтому использование фигурных скобок в данном случае обязательно.

В строке 7 написан оператор форматированного

вывода printf. Его первый аргумент, являющийся шаблоном вывода, будет выведен при вызове этого оператора, а спецификаторы типов %d будут заменены значением переменной count и значением элемента массива array со смещением count.

В строке 8 используется условие выбора. Выражение оператора if сравнивает значение count, изменяемое при каждой новой итерации цикла, со значением array_size, уменьшенное на 1. Далее следует функция форматированного вывода, содержащая только текстовый шаблон из двух символов: запятой и пробела. Текст в строке 8 выведет символы запятая и пробел четыре раза, для элементов array[0], array[1], array[2], array[3]. Назначение этой строки в правильном оформлении вывода программы. Функция printf стандартной библиотеки ввода-вывода использована по той причине, что функция не форматированного вывода puts вставляет после выводимых символов переход на новую строку, который в данном случае не требуется.

В строке 10 вызывается функция не форматированного вывода puts, выводящая точку.

В строке 11 закрывающая фигурная скобка завершает тело функции array_output.

Перейдем к рассмотрению исходного текста программы в файле 01.c, представленного на рисунке 3.

В строке 1 используется директива предпроцессора для включения в текст программы содержимого заголовочного файла стандартной библиотеки ввода-вывода. В тексте программы файла 01.c не используются функции ввода-вывода, строка 1 осталась там из-за того, что в начале написания примера планировалось разместить весь текст программы в одном файле, однако его размер оказался слишком велик и функции вывода были перенесены в отдельный файл, а директива

осталась. Данная ошибка никак не скажется на эффективности программы, т. к. любая библиотека включается в исполняемый файл только один раз, независимо от количества её подключений.

01.c | array_output.h | array_output.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include "array_output.h"

#define ARRAY_SIZE 5

int main () {
    int * array = NULL;
    array = malloc (sizeof(int)*5);
    int count = 0;

    // Отображение значений элементов массива
    array_output (array, ARRAY_SIZE);

    // Инициализация значений элементов массива
    for (count = 0; count < ARRAY_SIZE; count++)
        array[count] = 1;
    // memset((void *)array, 0, sizeof(array));

    array_output (array, ARRAY_SIZE);

    // Доступ к произвольным элементам массива
    array[2] = 2*count;
    array[ARRAY_SIZE-1] = ARRAY_SIZE;

    array_output (array, ARRAY_SIZE);

    free (array);
    return 0;
}
```

Рисунок 3 — Исходный текст программы, демонстрирующей основные операции с массивами

Однако лишние объявления необходимо избегать при использовании специальных библиотек, потому что может сложится ситуация, когда её функции нигде в

программе не используются, но от компилятора и компоновщика требуется добавление её кода в исполняемый файл. Невозможно гарантировать, что компоновщик правильно обработает эту ситуацию и не включит ненужную библиотеку.

Строка 2 содержит директиву предпроцессора, которая будет заменена содержимым файла stdlib.h из каталога include, поставляемого вместе с компилятором. Использования функций стандартной библиотеки не требует использования дополнительных ключей для компоновщика. В файле 01.c применяются функции malloc и обратная ей free.

В строке 3 содержится директива предпроцессора, которая будет заменена на содержимое файла memory.h. В этой библиотеке объявлена функция memset, которая закомментирована в тексте программы. Такая заготовка позволит продемонстрировать второй прием инициализации значений массива. Файл memory.h расположен в каталоге include и поставляется вместе с компилятором. Использование библиотеки memory не требует указания специальных ключей для компоновщика.

В строке 4 используется директива предпроцессора, предписывающая заменить эту строку содержимым файла array_output.h. Этот файл должен быть расположен в том же каталоге что и файл 01.c. На это указывают двойные кавычки, в которые заключено имя файла. Подключаемый заголовочный файл содержит всего одну функцию array_output, которая и используется в программе.

Строка 6 объявляет константу с идентификатором ARRAY_SIZE и значением 5. Данная константа соответствует количеству элементов в массиве. Объявление константы с помощью директивы

предпроцессора позволяет изменять размер массива и одновременно вносить все необходимые изменения для обработки массива нового размера, заменив цифру 5 в строке 6 на любое другое значение до 1000. В случае превышения 1000 повышается возможность разрушения программы в связи с переполнением стека.

В строке 8 содержится начало реализации функции main. Функция не принимает никаких значений, это видно из отсутствия символов между скобками и возвращает значение типа int. Возвращаемое значение будет передано в вызывающую программу, которой является ЭТ. В этой же строке содержится открывающая фигурная скобка, с которой начинается тело функции main. Закрывающая скобка содержится в строке 31.

В строке 9 объявляется и инициализируется переменная с идентификатором array. Переменная имеет тип указателя на целочисленный тип и инициируется значением NULL. NULL — означает указатель в никуда. Объявленный в этой строке указатель будет указывать на начало массива и все операции с элементами массива будут доступны только с помощью этого указателя. Если значение адреса, хранимое в переменной указателя будет изменено, будет потеряна возможность работы с элементами массива.

В строке 10 с помощью функции malloc выделяется память в области кучи размером, равным размеру переменной типа int умноженного на 5, т. е. под 5 переменных типа int. В случае успешного выделения памяти адрес начала выделенного участка памяти в области кучи присваивается значению указателя array. После выполнения этой строки, программа получит память для массива из общей свободной памяти ОЗУ вычислительной машины.

Строка 11 содержит объявление переменной типа int

с идентификатором count, инициализированную значением 0.

Строки 13, 16, 23 начинаются с символов // и являются комментариями. Все символы от последовательности // до символа конца строки пропускаются компилятором при обработке исходного текста программы.

В строках 14, 21, 27 вызывается функция array_output, реализованная в файле array_output.c. Её аргументами является указатель на начало массива array и количество элементов массива, переданное константой ARRAY_SIZE. После выполнения этой функции в стандартный поток вывода будут распечатаны значения элементов массива по указателю array.

Строка 17 содержит цикл с заданным количеством итераций, цикл for. Первое выражение обнуляет значение счетчика цикла count. Второе выражение является сравнением счетчика цикла count с условием прекращения выполнения цикла константой ARRAY_SIZE. Пока счетчик цикла меньше константы, будут выполнены операторы тела цикла. Третье выражение определяет шаг приращения счетчика цикла, который равен инкременту, т. е. увеличению на единицу. Тело этого цикла не ограничено фигурными скобками, поэтому телом может являться только одно выражение.

В строке 18 написано тело цикла for строки 17. Оно состоит из присвоения константного значения 1 элементу массива по указателю array со смещением count.

Строка 19 закомментирована. Она содержит вызов функции memset, в которую передается три аргумента: указатель на массив array, приведенный к типу указателя на void, т. е. указателю на переменную неизвестного типа. Внутри функции memset будет изменяться память

по этому указателю. Вторым аргументом является константное значение 0, которым будет заполняться память. Третий аргумент состоит из вызова оператора sizeof, возвращающий размер массива array.

Строки 17 и 18 изменят все значения массива на 1. Если закомментировать строки 17, 18 и разкомментировать строку 19, всем элементам массива будет задано значение 0. После изменения набора строк необходимо заново откомпилировать программу и после этого запустить.

В строке 24 производится изменение значения массива с индексом 2. Это будет третье значение массива, т. к. 2 — это смещение. После смещения на 2 элемента происходит доступ к третьему элементу. Элементу присваивается удвоенное значение счетчика цикла for, записанного в строках 17,18.

В строке 25 производится присвоение последнему элементу массива array со смещением 4 значения константы ARRAY_SIZE = 5.

В строке 29 выполняется освобождение памяти в области кучи, выделенной для массива по указателю array с помощью функции free.

В строке 30 оператор return возвращает значение 0 в вызывающую её программу, т. е. ЭТ. Под возвратом значения подразумевается помещение в регистр eax значения 0 и завершение работы программы.

Для создания исполняемого файла и его запуска использовалась комбинация команд из вызова компилятора gcc и запуска самой программы ./01 (Рисунок 4).

Исходный текст программы написан в трех файлах. Из них текст программы содержат два файла, поэтому для компилятора необходимо указать два файла с текстом программы array_output.c и 01.c. Сначало

указываются файлы с текстом функций, в конце файл с функцией main. Ключ -o позволяет задать имя исполняемого файла 01. Ключ -Wall предписывает компилятору все предупреждения считать ошибками. Таким образом уменьшается возможность допуска ошибки по невнимательности.

```
[valerii@]$ gcc array_output.c 01.c -o 01 -Wall - pedantic && ./01
array[0] = 0, array[1] = 0, array[2] = 0, array[3] = 0, array[4] = 0.
array[0] = 1, array[1] = 1, array[2] = 1, array[3] = 1, array[4] = 1.
array[0] = 1, array[1] = 1, array[2] = 10, array[3] = 1, array[4] = 5.
[valerii@]$
```

Рисунок 4 — Компиляция и исполнение программы, демонстрирующей основные операции с массивами

Ключ -pedantic предписывает выполнить проверку правильности исходного текста программы максимально строго, исключая всевозможные способы «додумывания» за программиста.

После запуска программы выведено три строки и работа программы была завершена. Все строки вывода сформированы вызовами функции array_output.

Первая строка сформирована вызовом в строке 14 файла 01.c. Вывод показал что все элементы массива имеют значение 0. В соответствии со стандартом языка программирования функция malloc не должна обнулять память. Фактически этого и не происходит. При создании массивов больших размеров, часть элементов массивов сразу после выделения памяти в области кучи будут иметь значения отличные, от нуля.

Вторая строка вывода сформирована вызовов функции в строке 21 файла 01.c. После проведения инициализации значений массива константным значением 1 в строках 17,18, все элементы будут равны

1.

Третья строка сформирована вызовом в строке 27 файла 1.c. Вывод третьей строки будет отличаться от второй измененными значениями элементов массива с индексом 2 (изменен в строке 24) и с индексом 4 (изменен в строке 25). В третьей строке значение с индексом 2 равно 10, а значение с индексом 4 равно 5.

Данный пример продемонстрировал неудобство совпадения названий файлов и функций. Несмотря на отсутствие жесткого требования к различиям, к правилам формирования имен файлам и функциям, лучше иметь фиксированное правило, которое позволит легко отличать, где используется название файла, а где функции или переменной.

Используемая в строке 19 функция memset может быть заменена другими функциями, например calloc — выделяющую и одновременно обнуляющую выделенную память. Обнулить уже выделенную память можно с помощью функции bzero, а изменить размер уже выделенной памяти с помощью функции realloc.

Подробное описание текста программы содержит также обсуждение различных ситуаций, которые могут произойти при изменении подхода к написанию текста программы или использовании другого приема.

В заключении раздела в качестве заметки следует отметить распространенный прием определения количества аргументов массива, используя операцию sizeof. Для этого используется выражение

int elements_count = sizeof array / sizeof array[0];

С помощью этого приема можно узнать количество элементов массива, зная только то, что он создан в области кучи и тип элемента массива. Работоспособность

этого приема следует проверять на целевой платформе. Не следует забывать, для работы со специализированными микропроцессорами компилятор языка Си может не полностью поддерживать стандарт языка программирования. Также может отличаться реализация стандартных библиотек.

4.1.3. Сортировка массива

Упорядочивание массива — это первый класс алгоритмов, который рассматривается при изучении массивов. Операции упорядочивания часто являются одним из начальных этапом обработки массивов данных. Исключение составляют задачи, в которых важна последовательность расположения значений в массиве. В таких случаях порядковый номер элемента массива является дополнительной размерностью и после увеличения размерности массива мерностью, содержащую порядковые номера исходного массива, к нему также становятся применимы алгоритмы упорядочивания.

Существует множество алгоритмов упорядочивания, каждый из которых наиболее эффективен в своей области. Под эффективность понимается минимизация времени выполнения или объема используемой памяти. Под областью применения, в свою очередь, понимаются виды функциональных зависимостей и прочие характеристики исходных наборов данных. Вопросы алгоритмизации, эффективности алгоритмов исследуются теорией алгоритмов. Одним из главных произведений в этой области для программистов является четырехтомник Дональда Кнута.

Считается, хороший специалист знает не только как применить некоторый алгоритм, но так же знает его ограничения, т. е. где его применять не эффективно или невозможно.

Рассмотрим два алгоритма сортировки qsort и пузырьковый метод. Алгоритм qsort состоит из двух частей: первая часть алгоритма ищет среднее значение среди значений элементов массива, вторая выполняет перестановку значений элементов массива таким

образом, чтобы значения большие или меньшие среднего оказывались по одну сторону от среднего. Далее производится разбиение двух полученных половин массива и выполняется перестановка элементов в каждой половине. Операция повторяется до полного упорядочивания всех значений массива. Вычислительная сложность этого алгоритма увеличивается незначительно при росте объема значений массива.

Второй алгоритм упорядочивания обычно знаком со школьной скамьи — пузырьковый метод. Его смысл заключается в сравнении двух ближайших значений и их перестановка в случае, если результат сравнения не удовлетворяет условию сортировки. Далее производится смещение на один элемент и сравнение повторяется. Вычислительная сложность этого алгоритма растет быстро, потому что в такой постановке алгоритма массив требуется обойти n^2 раз.

Сравнивая эффективность алгоритма qsort и пузырькового метода, нельзя однозначно сказать, какой из них будет эффективнее. Для примера будет выбран один из критических случаев, в которых алгоритм qsort не может демонстрировать своих преимуществ. Использован массив с малым количеством элементов, при этом значения элементов распределены не равномерно, и имеется значение, отличающееся от остальных на два порядка.

Пример исходного текста программы представлен на рисунке 1.

Исходный текст программы содержит функцию array_output, которая объявлена в файле array_output.h (пункт 4.1.2 Рисунок 1) и реализована в файле array_output.c (пункт 4.1.2 Рисунок 2).

Далее будет приведено построчное описание только тех строк исходного текста программы, для которых

ранее не давалось подробного описания.

01.c

```c
#include <stdio.h>
#include "array_output.h"

static int cmp(const void *p1, const void *p2) {
    const int *a = (int *)p1;
    const int *b = (int *)p2;
    return *a < *b ? *a : *b;
}

int main () {

    // Подготовка массива
    int array[] = {2, 1, 2, 1, 125};
    int array_size = 5;
    array_output (array, array_size);

    // Школьный пузырек
    int count = 0;
    for (count = 0; count < array_size; count++) {
        int j = 0;
        for (j = 0; j < array_size; j++) {
            if (array[count] < array[j]) {
                int temp = array[j];
                array[j] = array[count];
                array[count] = temp;
            }
        }
    }

    // QSort Указатель на массив, кол-во элементов, размер элемента, функция
    qsort(&array[0], array_size, sizeof(int), cmp);

    // Вывод результата
    array_output (array, array_size);

    return 0;
}
```

Рисунок 1 — Исходный текст программы упорядочивания массива различными методами

В строке 4 дано описание функции cmp, которая принимает два аргумента, имеющих тип указателя на константное значение пустого типа void. Таким образом

в функцию cmp передается два указателя на пустой тип. Тип void используется в случае, когда определить заранее тип значения не представляется возможным. Функция возвращает значение целого типа int. Спецификатор static предписывает заменить вызов функции cmp операторами, составляющими тело функции cmp. Тело функции состоит из операторов в строках 5, 6 и 7. В строке 8 закрывающая фигурная скобка завершает тело функции cmp.

В строке 5 объявлена переменная с идентификатором a, являющаяся указателем на значение, имеющее тип константы целого типа. В этой же строке производится инициализация переменной с идентификатором, a значением первого аргумента, переданного в функцию cmp и приведенного к типу указатель на целочисленный тип int.

В строке 6 объявлена переменная b, являющаяся указателем на константное значение целочисленного типа int, которое инициализировано значением второго аргумента функции cmp, приведенного к типу указатель на целочисленный тип int. Строки 5 и 6 отличаются только идентификаторами переменных, однако описание строк дано немного по разному, чтобы показать разные способы словесного описания текста программы.

Строка 7 состоит из вызова одного оператора return, который возвращает результат выражения, описанного в пункте 6.5.15 в стандарте языка программирования. Выражение производит сравнение двух значений, расположенных по адресу a и b. Для получения значений используется операция разыменования адреса, обозначаемая символом звездочка. В случае, если значение по адресу a меньше значения по адресу b, оператор return возвратит значение по адресу a, в случае, если значения по адресам a и b равны или

значение по адресу b меньше значения по адресу a, функция cmp вернет в точку своего вызова значение по адресу b.

В строке 13 объявляется массив с идентификатором array, каждый элемент которого имеет целочисленный тип int. При этом количество элементов массива не указывается, т. к. в квадратных скобках нет числа. В этой же строке массив инициируется значениями в фигурных скобках, перечисленных через запятую. Таким образом, после строки 13 будет доступен массив целочисленных значений, созданных на стеке и инициированных значениями 2, 1, 2, 1, 125.

В связи с тем, что количество элементов массива вычисляется компилятором, необходимо дополнительно создавать переменную или константу, содержащую количество элементов массива. Так как массив создан на стеке, количество его элементов меняться не может, поэтому более правильно было бы использовать для определения количества элементов массива константу. Объявление массива без указания количества элементов удобно использовать только при работе со строками, которые будут рассмотрены в следующих разделах.

В строке 14 объявляется переменная array_size и инициализируется значением, равным количеству элементов массива array.

В строке 15 и 34 вызывается функция вывода на экран значений элементов массива array_output, в которую в качестве аргументов передается указатель на массив array и количество элементов в нем.

Строки 17-28 реализуют алгоритм упорядочивания методом пузырька. Такое название он получил благодаря характеру изменений значений. В таблице 1 представлены значения элементов массива после каждой итерации внутреннего цикла for со счетчиком цикла j.

Максимальное значение 125 как бы «ныряет» в основание массива и затем всплывает, попутно расставляя в правильном порядке остальные значения элементов массива.

Таблица 1 — Изменение значений элементов массива array при пузырьковой сортировки

count	j	array[0]	array[1]	array[2]	array[3]	array[4]
0	0	2	1	2	1	125
0	1	2	1	2	1	125
0	2	2	1	2	1	125
0	3	2	1	2	1	125
0	4	125	1	2	1	2
1	0	1	125	2	1	2
1	1	1	125	2	1	2
1	2	1	125	2	1	2
1	3	1	125	2	1	2
1	4	1	125	2	1	2
2	0	1	125	2	1	2
2	1	1	2	125	1	2
2	2	1	2	125	1	2
2	3	1	2	125	1	2
2	4	1	2	125	1	2
3	0	1	2	125	1	2
3	1	1	1	125	2	2
3	2	1	1	2	125	2
3	3	1	1	2	125	2
3	4	1	1	2	125	2
4	0	1	1	2	125	2
4	1	1	1	2	125	2
4	2	1	1	2	125	2
4	3	1	1	2	2	125
4	4	1	1	2	2	125

В таблице 1 первый и второй столбец соответствует текущим значениям счетчиков циклов, как буд-то их вывод произведен оператором вывода между строками 26 и 27. Последующие столбцы содержат значения элементов массива после выполнения итерации внутреннего цикла. В таблице наглядно представлено, что в процессе сортировки происходит множество итераций, при которых значения не меняются. Алгоритм как бы «промахивается», выбирая элементы, расположение которых не будет изменено. Чтобы уменьшить количество таких «ошибочных» сравнений, полное правило пузырьковой сортировки включает в себя ограничение, гласящее, что сортировка прекращается, когда все элементы расставлены в соответствующем порядке.

Подробное построчное описание алгоритма затрудняет понимание принципа его работы, поэтому опишем принцип работы строк 17-28. Внешний цикл for со счетчиком цикла переменной count изменяет индекс элемента массива, со значением которого будут сравниваться все остальные элементы. Перебор сравниваемых элементов обеспечивает внутренний цикл for со счетчиком цикла j. Итерация внутреннего цикла выполняет перестановку элементов в случае, если элемент массива с индексом count меньше элемента массива с индексом j. Для перестановки требуется вспомогательная переменная temp, в которую временно перемещается одно из значений, во избежание его затирания. Аналогией для перестановки могут являться две корзины яблок. В одной корзине красные яблоки, в другой зеленые. Для того чтобы поменять яблоки в корзинах, необходима третья корзина, в которую на время нужно пересыпать яблоки, например, красные.

Затем из второй корзины в первую пересыпают зеленые яблоки, а из вспомогательной корзины во вторую пересыпают красные яблоки. В приведенном примере такой вспомогательной корзиной является переменная temp.

Следует отметить, в практической работе часто используются подобные таблицы для изучения работы неизвестной функции или библиотеки.

Строка 31 содержит вызов функции qsort, реализация которой содержится в стандартной библиотеке (stdlib). Текущая версия компилятора позволяет не проводить явного объявления стандартной библиотеки, т. к. она подключается по умолчанию. Функция qsort описана в пункте 7.22.5.2 стандарта языка программирования Си следующим образом

*qsort(void *base, size_t nmemb, size_t sizint (*compar) (const void *, const void *)*

Последний аргумент функции qsort — это указатель на функцию. Указанному прототипу соответствует функция программы cmp, указатель на которую передается в строке 31.

Для получения исполняемого файла из текста программы на рисунке 1, необходимо выполнить команду, приведенную на рисунке 2. Для успешной компиляции необходимо сначала указать файл с текстом функции array_output, а затем файл с исходным текстом программы 01.c. В процессе компиляции будет выдано предупреждение о неявной декларации функции qsort, т. к. она определена в файле stdlib.h, однако отсутствует директива include, предписывающая использовать заголовочный файл стандартной библиотеки. Предупреждение позволяет закончить создание

исполняемого файла.

```
[valerii@]$ gcc array_output.c 01.c
01.c: В функции «main»:
01.c:31:5: предупреждение: неявная декларация функции «qsort» [-Wimplicit-function-de
claration]
     qsort(&array[0], array_size, sizeof(int), cmp);
     ^
[valerii@]$
```

Рисунок 2 — Создание исполняемого файла

Для запуска программы необходимо выполнить команду запуска программы с именем по умолчанию.

./a.out

В результате запуска программы будет получен вывод, соответствующий выводу на рисунке 3.

```
array[0] = 2, array[1] = 1, array[2] = 2, array[3] = 1, array[4] = 125.
array[0] = 125, array[1] = 2, array[2] = 2, array[3] = 1, array[4] = 1.
```

Рисунок 3 — Результаты отображения стандартного потока вывода программы

Первая строка вывода сформирована функцией array_output, вызываемой в 15 строке. В этой строке значения в массиве располагаются так же, как и при инициализации массива в строке 13.

Вторая строка сформирована вызовом функции array_output, вызываемой в строке 34. До этого вызова массив сортируется в порядке возрастания пузырьковым методом и сортируется в порядке убывания функцией qsort.

Скорость выполнения кода описываемых реализаций алгоритмов на современном микропроцессоре настолько

велика, что для сравнения скорости их выполнения каждый из них был выполнен по 100 000 000 раз подряд. Для этого в исходный текст программы был добавлен еще один цикл с заданным количеством итераций. При каждом запуске один из методов сортировки был закомментирован. Время исполнения засекалось стандартной программой time с помощью вызова.

time ./a.out

Кроме того, был реализован полноценный пузырьковый алгоритм сортировки, завершающий работу в случае, если за полный проход внешнего цикла не осуществлялось ни одной перестановки значений.

Также были проведены замеры времени исполнения в разных направлениях, от большего к меньшему и наоборот. Такое исследование не выявило значительных отличий в продолжительности работы программы.

Напоследок для всех трех алгоритмов был изменен набор входных данных. На вход им был подан уже отсортированный массив. Численные результаты этих экспериментов представлены в таблице 2.

Таблица 2 — Сравнение производительности алгоритмов на разных наборах данных

	qsort	**Упрощенный пузырьковый метод**	**Пузырьковый метод**
Начальный набор	15,11	13,86	13,81
Отсортированный набор	15,11	13,85	0,65

Подведем итоги этого небольшого исследования реализаций алгоритмов. Как известно из теории

алгоритмизации, алгоритм qsort эффективнее пузырькового метода сортировки при обработке массивов значительного размера (более 1000 элементов). Скорость его выполнения замедляется медленно при увеличении количества элементов массива. Алгоритм qsort имеет одинаковую скорость работы для различных направлений сортировки, время его работы слабо зависимо от исходных данных.

Упрощенный пузырьковый метод может работать несколько быстрее, при очень малом количестве элементов в массиве, например менее 10. Он также слабозависим от характеристик входных данных. Однако анализируя реализацию его алгоритма хорошо видно, время его исполнения будет значительно увеличиваться при увеличении количества входных значений.

Полноценный пузырьковый метод имеет одно исключительное преимущество, чем меньше работы по сортировке необходимо произвести в массиве, тем быстрее алгоритм обрабатывает массив. А если входной массив почти или полностью отсортирован, время его исполнения в десятки раз меньше, чем у упрощенного пузырькового алгоритма.

Приведенное в этом пункте описание сравнения алгоритмов сортировки может являться примером обсуждения сложных вопросов со старшими товарищами, лидером группы или ведущим разработчиком. Обычно в результате его не даются готовые решения проблем, стоящих перед программистом, а предлагаются способы поиска решения, содержащие устные рекомендации и примеры. В связи с этим, умение писать исходный текст программы со слов товарищей, является обязательным навыком даже для начинающего программиста.

Этим примером автор хочет обратить внимание на то, что наличие вычислительной машины не избавляет от

необходимости думать, размышлять. Также, крайне желательно, для каждой задачи разрабатывать персональное математическое решение, что может её ускорить в десятки раз.

4.2. Составные типы данных: структуры и объединения

4.2.1. Объединения

Объединения используются в двух случаях.

1. Требуется сэкономить память ОЗУ за счет размещения нескольких переменных в одних ячейках памяти. Эти переменные не должны использоваться одновременно. А после каждой смены роли экземпляра объединения необходимо задавать актуальное значение еще раз.

2. Требуется интерпретировать один участок ОЗУ по разному или изменить порядок следования байт.

Объединения описаны в стандарте языка программирования в разделе 6.7.2.1 Спецификаторы структур и объединений. Элементами объединения могут быть значения любых, в том числе составных, типов данных. Однако использование объединений в некоторых случаях могут привести к неопределенным результатам, поэтому необходимо ознакомится с указанным разделом стандарта.

Размер объединения в ОЗУ равен размеру самого большого типа данных, включенных в него.

```
union utr {
   unsigned int value;
   char bytes[2];
};
```

Объединение utr может интерпретироваться как переменная беззнакового целого типа с идентификатором value или как массив с идентификатором bytes с двумя элементами, каждый из которых имеет тип char. Для хранения переменной типа

unsigned int на ПЭВМ с использованной версией ОС и компилятора необходимо 4 байта. Для переменной bytes требуется 2 байта. Размер объединения определяется по максимальному значению, поэтому размер будет равен 4 байтам. Проверить размер объединения можно с помощью оператора sizeof.

printf("%ld\n\n", sizeof (union utr));

В примере вызывается функция форматированного вывода printf, в шаблоне вывода которой указана переменная типа long int и специальная последовательность символов перехода на новую строку. Вторым аргументом вызывается оператор sizeof. В качестве аргумента для оператора sizeof передается тип объединения utr.

На рисунке 1 приведен исходный текст программы определения аппаратной архитектуры ПЭВМ big ending или little ending. Имеется в виду способ расположения данных. Big ending условно означает, что разряд, отвечающий за тысячи и сотни в числе находятся в ячейках ОЗУ с большими значениями адресов, по сравнению с разрядом, отвечающим за десятки и единицы. Соответственно, условно, в архитектуре big ending при чтении в направлении роста адресов, т. е. СПРАВА НАЛЕВО, сначала расположены единицы, потом десятки. В архитектуре little ending наоборот, сначала расположены десятки, затем единицы.

При обработке файла 01.c первая строка будет заменена содержимым файла stdio.h . В программе используется функция форматированного вывода стандартной библиотеки ввода/вывода.

Вторая строка будет заменена содержимым файла string.h . В программе используется функция bzero из

библиотеки string, которая обнуляет память.

01.c

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

union utr2 {
    int value;
    char bytes[4];
};

int big_end (void) {
    union utr2 variable;
    variable.value = 1;
    variable.bytes[0] = 0;
    return variable.value?1:0;
}

int main () {
    union utr2 variable;
    printf("%ld\n", sizeof (variable));
    printf("%ld\n\n", sizeof (union utr2));
    bzero ((void *) &variable, sizeof (variable));
    printf("%s\n\n", big_end()?"да":"нет");
    return 0;
}
```

Рисунок 1 — Исходный текст программы определения аппаратной архитектуры ПЭВМ big ending или little ending

Строки 4-7 содержат объявление объединения utr2. Данное объединение может использоваться как целочисленная переменная типа int с идентификатором value, или как массив из четырех элементов типа char с идентификатором bytes.

Строки 9-14 содержат объявление и реализацию функции big_end, которая не принимает аргументы и возвращает значение типа int. В строке 10 объявляется переменная с идентификатором variable типа union str2. После выполнения этой строки переменная variable будет

содержать мусорное значение. В строке 11 объединение variable используется как переменная типа int, которому присваивается значение 1. Таким образом изменяются все байты объединения variable. Значение 1 состоит из одного бита, равного 1, остальные биты должны быть установлены в 0. Для установления архитектуры big ending необходимо определить, в каком байте находится бит, равный 1.

В строке 12 объединение интерпретируется как массив байтов bytes. Байт с нулевым смещением от начала массива затирается значением 0. Таким образом, если единица в числе находилась раньше значений десятков и сотен, т. е. используется архитектура big ending, единица не будет затерта нулем. Если сначала находятся сотни, в конце единицы, число изменится и value будет равно 0.

В строке 13 оператор return завершает выполнение функции big_end и возвращает инвертированное значение variable.value , т. е. если variable.value = 1, функция возвращает 0 и наоборот.

Строка 17 соответствует строке 10.

Строка 18 выведет размер переменной variable типа union utr2.

Строка 19 выведет размер типа union utr2, который должен быть равен предыдущему выводу.

В строке 20 используется функция bzero библиотеки string, обнуляющая память по переданному в неё указателю размером, равным второму аргументу функции. Вызов данной функции демонстрирует возможность работы с объединениями с помощью операторов работы с памятью, однако использовать такие функции для обработки массивов объединений с высокой степенью вероятности приведет к ошибкам в программе.

В строке 21 вызывается функция printf. В её первом

аргументе, шаблоне вывода, указан спецификатор типа строка %s, который будет рассмотрен в следующих разделах. Вторым аргументом является вызов конструкции, которая в качестве выражения использует результат вызова функции big_end. Если результат работы функции big_end равен 1, то будет выведена первая строка со словом «да», если равен 0, со словом «нет».

Для создания исполняемого файла и его запуска использовалась комбинированная команда, приведенная на рисунке 2.

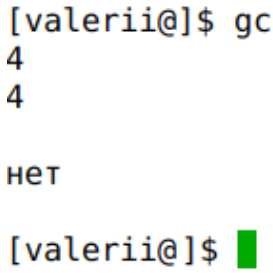


Рисунок 2 — Создание исполняемого файла и его исполнение

Первая строка вывода сформирована 18 строкой исходного текста программы. Вторая строка и пустая третья сформированы 19 строкой исходного текста. Слово «нет» и четвертая пустая строка выведена 21 строкой исходного текста программы.

4.2.2. Структуры

Структуры используются для объединения значений разных типов в один тип. Таким образом повышается читаемость исходных текстов программы и появляется возможность выполнять часть действий с помощью функций для работы с памятью. Основное отличие от массивов заключается в том, что элементы структуры располагаются не всегда последовательно друг за другом. Часто, для повышения скорости выполнения программы, при создании экземпляров структур компилятор добавляет смещение после переменной. Такое расположение в ОЗУ позволяет ускорить работу программы, но может привести к перерасходу ОЗУ и вызывать ошибки при использовании функций по работе с памятью.

Структуры описаны в стандарте языка программирования в разделе 6.7.2.1 Спецификаторы структур и объединений. Способы их инициализации в разделе 6.7.9.

Далее по тексту дано пояснения только по строкам, имеющим отношение к работе со структурами.

В строках 4-7 объявлен тип именованная структура с идентификатором str2, состоящая из двух элементов. Переменная value имеет тип int, переменная bytes является массивом из 4 элементов, каждый из которых имеет тип char.

В строках 9-13 объявлен тип именованная структура str1, которая включает в себя переменную dbl типа double, переменную ch типа char и переменную elem_str2 являющуюся структурой типа str2.

В строках 16-17 показан синтаксис инициализации элементов структур. Следует обратить внимание на два приема. Во-первых, инициализацию можно проводить не

по порядку следования элементов при объявлении структуры. Во-вторых, во время инициализации структуры можно задавать значения для вложенных структур и массивов.

01.c

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

struct str2 {
    int value;
    char bytes[4];
};

struct str1 {
    double dbl;
    char ch;
    struct str2 elem_str2;
};

int main () {
    struct str1 variable = {.ch = 11, .elem_str2.value = 1,
                            .elem_str2.bytes[0] = 0};
    printf("%ld\n", sizeof (struct str2));
    printf("%ld\n", sizeof (variable));
    printf("%ld\n\n", sizeof (struct str1));

    printf("%ld\n", offsetof(struct str1, ch));
    printf("%ld\n\n", offsetof(struct str1, elem_str2.bytes[3]));

    variable.elem_str2.bytes[3] = 10;
    printf("%d\n", variable.elem_str2.bytes[3]);
    printf("%d\n", variable.elem_str2.value);
    return 0;
}
```

Рисунок 1 — Исходный текст программы, демонстрирующей выполнение операций со структурами и её элементами

В строках 18-20 написаны строки, которые приведут к выводу размеров структур в байтах.

В строке 22 использована функция

форматированного вывода printf, одним из аргументов которой является макрокомманда offsetof, возвращающая смещение в байтах до элемента от начала структуры. Таким образом в строке 22 будет выведено смещение до элемента ch структуры struct str1.

В строке 23 будет выведено смещение от начала структуры struct str1 до последнего элемента в структуре str1.elem_str2.bytes[3].

В строке 25 приведен пример задания значения элемента структуры. Значение 10 задается последнему элементу массиву bytes, который является вложенным в структуру типа struct str2, которая, в свою очередь, является элементом структуры типа struct str 1.

В строках 26-27 показан прием обращения к элементам структур в функции форматированного вывода printf.

```
[valerii@]$ gcc 01.c && ./a.out
8
24
24

8
19

10
1
[valerii@]$
```

Рисунок 2 — Создание исполняемого файла и его исполнение

Создание исполняемого файла и его запуск выполнены одной командой (Рисунок 2).

Первая строка вывода сформирована выполнением 18 строки исходного текста программы. Цифра 8 — это размер в байтах struct str2. Структура данного типа содержит те же элементы, что и объединение в пункте

4.2.1, однако объединение занимает в ОЗУ 4 байта, а структура 8 байт.

Вторая и третья строки вывода сформированы 19 и 20 строками текста программы. Значение 24 соответствует размеру структуры типа struct str1. В строке 19 структура передается с помощью идентификатора переменной, а в строке 20 как тип структуры.

Четвертая пустая строка сформирована второй служебной последовательностью перехода на новую строку первого аргумента оператора printf в строке 20.

Значение 8 строки 5 сформировано строкой 22 текста программы и означает, что элемент ch структуры struct str1 смещен на 8 байт от начала структуры.

Значение 19 строки 6 сформировано строкой 23 текста программы. Получается, последний элемент структуры struct str1 расположен в 19 байте, однако размер структуры равен 24 байтам. Объяснение такого расхождения произведено в следующем разделе книги.

Седьмая строка не содержит символов.

Восьмая строка вывода содержит значение элемента str1.elem_str2.bytes[3] .

Девятая строка содержит значение элемента str1.elem_str2.value .

Строки вывода 8 и 9 подтверждают, что значение, хранимое в структуре, не перезаписывает другие элементы структуры.

При использовании структур необходимо отслеживать смещения элементов от начала структуры, с целью определения фактического выравнивания элементов структур на данной ЭВМ, конкретной ОС и версии компилятора. В остальном работа с элементами структур производится так же, как и с элементами массивов.

4.2.3 Расположение элементов структур в ОЗУ ЭВМ

Как было показано в предыдущем разделе, размер памяти, требуемой для размещения структуры, редко совпадает с тем, который получается суммированием размеров памяти её элементов. Причиной этого являются правила работы микропроцессоров с выравниванием данных. Использование выравнивания микропроцессором позволяет выполнять операции с данными одной инструкцией микропроцессора, а обработка данных без выравнивания требует двух или более инструкций.

Проблема выравнивания объектов возникает при использовании функций, оперирующих участками памяти, адресной арифметики или при обработке данных от аппаратных устройств. Использование операций с памятью или приемов применения адресной арифметики при неправильно вычисленном размере структуры превращают данные в мусор, а наибольшие потери памяти ОЗУ возникают при использовании массивов структур. В этом случае, потери могут достигать половины всей потребляемой программой памяти. Также возможно разрушение памяти программы из-за выхода за пределы области данных, получения мусорного указателя из полей структуры и по другим причинам.

Само по себе **правило выравнивания одно**. Объект каждого типа должен начинаться с адреса, кратного размеру своего типа. Например:

- char имеет размер один байт, поэтому может начинаться с байта с любым номером;
- если short занимает 2 байта, то он должен начинаться с байтов с номером, кратным двум;
- если int или float занимают по 4 байта, значит должны начинаться с адреса, кратного 4;

- если указатель на int занимает 8 байт, то должен начинаться с адреса, кратного 8 и т.д..

Язык программирования Си не предписывает компилятору контролировать расположение в памяти элементов структур. Подобное правило отсутствует с целью сохранения максимальной совместимости с аппаратным обеспечением различных архитектур.

Неиспользуемые байты появляются в тех структурах, в которых сначала расположены данные большего размера, а затем данные меньшего размера. Например, вначале char , затем double. Другой причиной является использование компилятором байт после структуры, для выравнивания начала следующей структуры.

Решение проблемы выравнивания заключается в расположении в начале структуры наиболее объемных данных и далее по уменьшению их размера, например: int , int затем char. Такой подход к решению проблемы выравнивания не всегда применим, т. к. он понижает читаемость исходного текста программы.

При обработке данных аппаратных интерфейсов часто используется директива #pragma pack, отключающая выравнивание для структуры. Такой пример будет рассмотрен в данном разделе. Использование этой директивы гарантирует отсутствие выравнивания структур, однако может замедлять работу программы. Для микропроцессоров архитектуры x86 и x86-64 замедление скорости выполнения программы обычно незначительно или отсутствует вовсе.

Первая и вторая строки содержат директивы предпроцессора и имена заголовочных файлов, содержащих объявления функций форматированного ввода вывода и задания значения для участка ОЗУ.

В строках 4-6 содержится объявление типа именованной структуры str1. Её элементы расположены

с нарушением правила выравнивания элементов структуры. Элементы структуры набраны в текстовом файле в одну строку с нарушением правил оформления структур, в соответствии с которым каждое выражение должно быть набрано в отдельной строке. В данном случае это оправдывается тем, что правильное оформление заголовочного файла увеличило бы количество строк почти в два раза.

01.c 01.h

```
1   #include <stdio.h>
2   #include <string.h>
3
4   struct str1 {
5       double dbl1;    char ch1;    int int1;    short sh1;
6   };
7
8   struct str1_small {
9       char ch1;       short sh1;   int int1;    double dbl1;
10  };
11
12  #pragma pack(push, 1)
13  struct str1_opt {
14      double dbl1;    char ch1;    int int1;    short sh1;
15  };
16  #pragma pack(pop)
17
18  union un1 {
19      struct str1 struct1;          char bytes[24];
20  };
21
22  union un1_small {
23      struct str1_small struct1;    char bytes[24];
24  };
25
26  union un1_opt {
27      struct str1_opt struct1;      char bytes[24];
28  };
29
```

Рисунок 1 — Исходный текст заголовочного файла с объявлениями структур и объединений

В строках 8-10 объявлен тип структуры str1_small, в

которой элементы расположены обратно правилу выравнивания, т. е. по возрастанию размера.

В строках 12-16 приведено объявление str1_opt, заключенное в директивы препроцессора, отключающие выравнивание. Следует обратить особое внимание на наличие закрывающей директивы, включающей правила выравнивания после завершения объявления типа данных str1_opt.

В строках 18-20 объявлен тип объединения un1, в который включена структура struct1 типа str1 и массив из 24 элементов типа char. Соответственно структура str1 и массив bytes будут использовать одну и ту же область памяти ОЗУ.

В строках 22-24 объявлен тип объединения un1_small, отличающийся от объединения un1 использованием структуры типа str1_small .

В строках 26-28 объявлен тип объединения un1_opt, отличающийся от un1 использованием структуры типа str1_opt.

Значение размера массива bytes в 24 элемента получен предварительно по размеру структуры, которая использует память максимально неэффективно из описанных.

Далее приведено построчное описание текста программы файла 01.c, представленной на рисунке 2.

Строка 1 обрабатывается предпроцессором, который заменяет её на содержимое файла 01.h, находящегося в этом же каталоге.

Строка 2 предписывает предпроцессору выполнить замену набора символов ARRAY_SIZE на константу 24 по всему тексту файла 01.c.

Строки 4-8 содержат объявление и реализацию функции print_array. Функция принимает один аргумент типа указатель на символьный тип char. Функция не

возвращает никаких значений, на что указывает ключевое слово void перед названием функции.

Строка 5 содержит объявление переменной с идентификатором count и инициализацию её константным значением 0.

Строка 6 включает в себя цикл while с условием выполнения тела этого цикла, которое состоит из вызова оператора printf и операции инкремента переменной count. Тело цикла while состоит из двух выражений и не содержит обрамляющих фигурных скобок, т. к. они объединены операцией следования, обозначающейся запятой. Выражения можно объединять операцией следования только в том случае, если при их исполнении не требуется изменять состояния стека программы. Имеется ввиду, что значения в стеке могут изменяться, но операции, объединяемые запятой, не должны создавать новых переменных на стеке или удалять их.

Следует обратить внимание, в строке 6 все операторы цикла написаны в одну строку, что можно рассматривать как нарушение стиля программирования. В данном случае это сделано с целью уменьшения размера изображения с исходным текстом.

В строке 7 содержится оператор puts, который выводит пустую строку. Таким образом создается визуальный отступ между отображениями значений элементов массива. Этот оператор улучшает читаемость информации, выводимой в окне ЭТ, и не несет смысловой нагрузки с точки зрения алгоритма функции.

В строке 10 определен используемый прототип функции main. В данном случае она не принимает и не возвращает никаких значений.

Строки 11-13 содержат операторы форматированного вывода printf, вторым аргументом которых является операция sizeof, возвращающая размер структур типов:

struct str1, struct str1_small и struct str1_opt.

01.c 01.h

```
1  #include "01.h"
2  #define ARRAY_SIZE 24
3
4  void print_array (char * array) {
5      int count = 0;
6      while (count < ARRAY_SIZE) printf("%d ", array[count]), count++;
7      puts("");
8  }
9
10 int main () {
11     printf ("%ld\n", sizeof(struct str1));
12     printf ("%ld\n", sizeof(struct str1_small));
13     printf ("%ld\n", sizeof(struct str1_opt));
14     union un1 var1;
15     union un1_small var1_small;
16     union un1_opt var1_opt;
17     memset (&var1,       255, ARRAY_SIZE);
18     memset (&var1_small, 255, ARRAY_SIZE);
19     memset (&var1_opt,   255, ARRAY_SIZE);
20     var1.struct1.dbl1 = 1.0;
21     var1.struct1.ch1 = 1;
22     var1.struct1.int1 = 1;
23     var1.struct1.sh1 = 1;
24     var1_small.struct1.dbl1 = 1.0;
25     var1_small.struct1.ch1 = 1;
26     var1_small.struct1.int1 = 1;
27     var1_small.struct1.sh1 = 1;
28     var1_opt.struct1.dbl1 = 1.0;
29     var1_opt.struct1.ch1 = 1;
30     var1_opt.struct1.int1 = 1;
31     var1_opt.struct1.sh1 = 1;
32     print_array (&var1.bytes[0]);
33     print_array (&var1_small.bytes[0]);
34     print_array (&var1_opt.bytes[0]);
35     return 0;
36 }
37
```

Рисунок 2 — Исходный текст программы, демонстрирующей размещение элементов структур в ОЗУ

В строках 14-16 объявлены объединения union un1 var1, union un1_small var1_small и union un1_opt var1_opt.

С их помощью будет выведено содержимое ячеек ОЗУ, в которых находятся структуры, указанные ранее.

В строках 17-19 применена функция memset. Каждому байту объединений будет задано значение 255, которое в двоичном виде записывается как 11111111, значит все биты объединений будут равны 1.

В строке 20 объединение var1 трактуется как структура типа struct1. Элементу dbl1 типа double структуры struct1 присваивается константное значение 1.0. При задании константного значения для переменных типов с плавающей запятой необходимо не забывать указывать значение дробной части числа.

Присвоение строки 20 заменит биты с единицами объединения var1 на соответствующие константе 1 типа double. В строках 21-31 задаются значения всем элементам объединений, интерпретируемым как структуры.

В строках 32-34 производится вызов функции print_array. В функцию передаются указатели на объединения, как массивы, а не структуры. Таким образом, при каждом вызове print_array будут выведены значения всех байт, занимаемых структурой, переданной по указателю.

Строка 35 содержит вызов оператора return, завершающего выполнения функции main, а значит и всей программы.

В строке 36 закрывающая фигурная скобка ограничивает функцию main.

На рисунке 3 представлен результат выполнения текста программы, представленной на рисунке 2.

Для создания исполняемого файла применена комбинация вызова программы gcc с передачей в качестве параметра имени файла 01.c операции && и вызова созданного исполняемого файла с именем по-

умолчанию a.out, расположенном в текущем каталоге.

```
[valerii@]$ gcc 01.c && ./a.out
24
16
15
0 0 0 0 0 0 -16 63 1 -1 -1 -1 1 0 0 0 1 0 -1 -1 -1 -1 -1 -1
1 -1 1 0 1 0 0 0 0 0 0 0 0 0 -16 63 -1 -1 -1 -1 -1 -1 -1 -1
0 0 0 0 0 0 -16 63 1 1 0 0 0 1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1
[valerii@]$
```

Рисунок 3 — Создание исполняемого файла и его исполнение

Значение 24 соответствует размеру структуры struct str1 в байтах. При расположении элементов структуры в соответствии с правилом выравнивания, эти же данные будут занимать в ОЗУ 16 байт, как структура типа str1_small. Это подтверждает вывод второй строки в ЭТ. Упаковка структуры в памяти позволяет сохранить эти же значения, используя 15 байт, что демонстрируется выводом третьей строки.

Далее побайтово выводятся значения объединений, в которые включены рассматриваемые типы структур.

Чтобы понять вывод программы в строках 4-6, следует обратить внимание на следующие факты:

- в файле 01.c в строке 6 в первом аргументе функции printf указан спецификатор типа %d, т. е. планируется вывод значений целого типа int;

- в функцию print_array передается указатель на массив, каждый элемент которого имеет тип char (знаковый символ);

- если все биты значения типа char равны 1, это соответствует значению -1 в дополнительном коде (см. пункт 1.2);

- значение -1 типа char, переданное в оператор

форматированного вывода, в котором для этого значения указан спецификатор целого числа, будет выведен как целое число, т. е. будет равно -1;

- в используемом процессоре архитектуры x86-64 применяется архитектура Big ending, а значит сначала хранятся байты с меньшими номерами, затем с большими, т. е. в порядке, обратном записи человеком на бумаге. Например, единица записывается так: 1 0 0 0 0 0 0 0.

При использовании структуры типа struct str1, которая соответствует четвертой строке вывода в ЭТ, в памяти сохранилось четыре значения -1, а значит четыре байта используются для выравнивания внутри структуры. Значения -1 в конце строки соответствуют неиспользованным байтам.

При использовании структуры типа struct str1_small, которая соответствует пятой строке вывода в ЭТ, в памяти сохранилось всего одно значение -1, а значит только один байта используется для выравнивания внутри структуры. Кроме того, сама структура «короче» в памяти на 24-16=8 байт. Значения -1 в конце строки соответствуют неиспользованным байтам. Таким образом, применение правила выравнивания позволило сэкономить 8 байт.

При использовании структуры типа struct str1_opt, которая соответствует шестой строке вывода в ЭТ, значения -1 в памяти не сохранилось. Байты для выравнивания внутри структуры не использовались. Упакованная структура имеет самый маленький размер в памяти, 15 байт. Значения -1 в конце строки соответствуют неиспользованным байтам.

Для работы с выравниванием используется два макроса: alignas, предписывающий выполнить выравнивание в соответствии с переданным в него

значением и alignof, отображающий смещение элемента от начала структуры.

Значение возвращаемое макросом alignof для базовых типов совпадает с размером этого типа, т. е. вызовом оператора sizeof.

Например:

alignof(struct str1);

будет равен 8.

alignof(struct str1_small);

будет равен 8.

alignof(struct str1_opt);

будет равен 1.

Макрос alignas предписывает выровнять по переданному значению, т. е. разместить с адреса, кратного переданному значению, но не предписывает, где именно разместить эти данные. В связи с этим выравненные переменные не обязательно будут располагаться друг за другом и могут находиться в разных участках памяти.

Пример:

alignas(8) struct str1 variable;

Расположить структуру variable с адреса, кратного 8.

Знания только стандарта программирования не позволяют писать правильные исходные тексты

программ. В стандарте специально не выполнена жесткая регламентация расположения в памяти структур. Это сделано для поддержки максимально возможного количества аппаратных и программных платформ.

Написание правильно работающих текстов связано не со стандартом, а с его реализацией на разных платформах. Настоятельно рекомендуется проверять работу приемом выравнивания на конкретной аппаратной и программной архитектуре во избежания получения скрытых ошибок в исполняемом файле.

Для эффективного использования приемов выравнивания данных сложных типов требуются знания в области аппаратного обеспечения. Другим подходом является экспериментальная проверка размеров структур, адресов расположения всех элементов в них. Также рекомендуется выносить все аппаратнозависимые объявления в отдельный файл.

4.3. Составные типы данных: строки
4.3.1. Объявление и инициализация строк

Для обработки текстовых данных используются одномерные массивы. Отличие от обычных массивов состоит в наличие нулевого символа в конце строк. Он необходим по причине отсутствия информации о длине строки. Алгоритмы всех стандартных функций для обработки строк работают корректно только при наличии в конце строки нулевого символа. Ответственность за наличие конечного нулевого символа лежит на программисте, хотя стандартные функции облегчают его работу, т. к. дописывают нулевой символ при выполнении своих алгоритмов.

Одной из главных проблем при работе со строками является правильная конвертация текста из одной кодировки в другую. Кодировки накладывают ограничения не только на операции со строками. Их также необходимо учитывать при задании константных значений в исходном тексте программы.

Изначально использовалась однобайтовая кодировка текстовых символов ASCII. В тот период появилась библиотека для работы со строками, функции которой описаны в файле string.h. Кодировка ASCII состоит из двух частей: базовой и расширенной части. Базовая часть включает в себя управляющие команды, символы английского алфавита и цифры. Расширенная часть включает символы национальных алфавитов и символы псевдографики и может меняться в зависимости от кодировки.

Наиболее популярными кодировками для латиницы стали koi8-r (в POSIX системах), CP866 (сейчас используется в основном в ОС DOS и ЭТ ОС семейства WindowsTM) и windows-1251 (в графическом интерфейсе

ОС семейства WindowsTM и сайтах оптимизированных под браузеры фирмы MicrosoftTM). Соответственно, константы, заданные для кодировки koi8-r, будут иметь другое значение в кодировках CP866 и windows-1251.

Следующим этапом стало появление кодировки с переменным количеством байт на символ. Наибольшую популярность в сети Интернет приобрела кодировка utf-8, т. к. она позволяет передавать управляющие команды сетевых протоколов также эффективно, как и с помощью ASCII, а также использовать до четырех байт для передачи текстовых и прочих символов.

Возможно использование и двухбайтовой кодировки utf-16, в которой каждый символ кодируется не менее чем двумя байтами.

Такое обилие кодировок иногда приводит к мысли о создании собственного внутреннего представления текстовых данных с написанием функций преобразования во внутренний формат и из него. Следует учитывать, внутренний формат не сможет хранить все символы кодировок utf-8 и utf-16, если его разрядность менее двух байт. Однако основной проблемой для «собственного» формата текстовых данных станет использование систем управления базами данных СУБД, которые используют стандартные кодировки. Также может не быть возможности применения библиотечных функций для работы со строками.

Описание задания текстовых констант приведено в разделе 6.4.5 стандарта языка программирования Си. Для хранения текста в разных кодировках используются различных типы данных:

- для хранения символов ASCII используется тип char (префикс не используется);

- для хранения символов utf-8 используется тип int

(префикс u8);

- для хранения символов из нескольких байт (wide string literals) используются следующие типы:
  - тип wchar_t (префикс L);
  - тип char16_t или char32_t (префикс u или U).

Для преобразования текста из текста соответствующего текущей локальной настройки (многобайтовой строки) в формат строки широких символов:

- wchar_t используется функция mbstowcs;
- char16_t используется функция mbrtoc16;
- char32_t используется функция mbrtoc32.

Использование того или иного формата строк широких символов связано с особенностями аппаратной архитектуры, установленного программного обеспечения или конфигурации конкретного устройства. Также причиной выбора одного из типов может стать наличие готовых функций обработки текстовых данных.

Следует обратить особое внимание на тип wchar_t. Он использует различное количество байт для хранения одного символа в ОС WindowsTM (равен 16 бит) и POSIX (соответствует int).

Приведем примеры инициализации строк ASCII, в которых каждый символ занимает один байт.

```
char ch[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

соответствует

```
char ch[] = {'h', 'e', 'l', 'l', 'o', 0};
```

соответствует

```
char ch[] = "hello";
```

соответствует

```
char ch[6] = "hello";
```

компилятор сам рассчитывает количество символов в строке с учетом завершающего нулевого символа.

```
char ch[] = "";
```

указанная строка создает массив из одного элемента, являющегося нулевым символом.

Другим типом данных описываются символы текста. Отличием при инициализации символов текста является синтаксис и возможность инициализации только одного значения.

```
char ch = 'h';
```

Данная строка инициализирует переменную из одного символа. Нулевой символ не может быть добавлен по определению, т. к. эта переменная не является массивом.

Символы подлежат сравнению между собой с помощью оператора условного перехода if, т. к. они являются целочисленными значениями типа char. В свою очередь, для сравнения строк необходимо использовать специальные функции, т. к. необходимо выполнить сравнение двух последовательностей.

Некоторым парадоксом для начинающего программиста кажется необходимость «угадывать», какое количество символов может потребоваться хранить в этой переменной за все время выполнения программы.

Такой поход пошел с тех времен, когда текст состоял из заранее определенных наборов символов и переносился из проектной документации в исходный текст программы. Обычно поле вывода имеет фиксированную ширину и не позволяет выводить бесконечный текст. Отчасти эта проблема решается использованием динамически выделяемой памяти в области кучи.

Многобайтовые константные значения допустимо задавать с помощью числовых кодов, например так

```
L'\1234'
```

В этом случае строка **обязательно** должна иметь соответствующий её комментарий, в котором приводится пояснение значения константы. В среде программистов константы, значения которых не объяснены, называются «магическими» значениями. В том смысле, что все их видят, но не кто не знает, как они работают.

В соответствии со стандартом языка программирования, если один из символов указывается как широкий символ, все остальные символы этой константной строки считаются широкими символами.

Например:

```
"a" "b" L"c"
```

эквивалентно

```
L"abc"
```

Смешение же символов разного количества байт приведет к «неопределенному поведению».

В разделе 6.4.3 стандарта языка программирования указана возможность задания значений широких

символов с помощью следующей комбинации символов

"\x12"

или

ch = \u0192

В разделе 7.30 стандарта языка программирования выполнена классификация типов широких символов и функции отображения между разными типами.

В разделе 7.30.1 определены типы wint_t, wctrans_t, wctype_t, а также макроопределение WEOF, используемое для определения конца файла, содержащего многобайтовые строки. Значение LC_TYPE позволяет определить текущие локальные настройки.

В следующем разделе этой книги будут рассмотрены функции отображения данных между разными типами.

4.3.2 Функции для работы со строками

Освоение функций для работы со строками удобнее проводить с использованием однобайтовых ASCII строк. Минимальный набор для обработки строк состоит из пяти функций: strlen, strcpy, strcat, ctrcmp и strstr. Они определены в заголовочном файле string.h, который необходимо подключать директивой include. Библиотека, соответствующая этому заголовочному файлу, подключается компоновщиком по умолчанию, поэтому нет необходимости указывать ключи для использования этих функций.

Также необходимо использовать функции конвертирования текстовых строк в цифры, даты, прочие форматы и обратно. За преобразование текста в целое число отвечают функции atoi, atol и atoll, описанные в заголовочном файле stdlib.h. Для преобразования текста в число с плавающей запятой используются функции atof, atol. Передаваемый в эти функции текст должен завершаться нулевым символом.

Для преобразования символов внутри строки большего размера используются функции strtod, strtof, strtold. Они используют два аргумента, на начало и конец фрагмента строки, который необходимо перевести в число. В этом случае фрагмент строки не должен заканчиваться нулевым символом. Данные функции также описаны в заголовочном файле stdlib.h.

Для преобразования цифр и чисел в текст используется функция sprintf, которая преобразует цифры в текст в соответствии с переданным в ней шаблоном вывода и спецификаторам вывода. Для корректной работы функции необходимо передавать указатель на массив, в который будут выведены преобразованные числа. Функция asprintf не только

выполняет преобразование по примеру sprintf, но и корректно выделяет память под сформированную строку.

Используя функции fprintf, fscanf, описанные в заголовочном файле stdio.h, есть возможность выполнять преобразование текста в числа и наоборот налету при получении или записи данных в поток. Этот прием позволяет выполнить преобразование без использования промежуточных буферов, объединив операции чтения и записи в поток с преобразованием данных в нужный формат.

Таким образом, функции для работы со строками присутствуют в библиотеках, описанных в заголовочных файлах string.h, stdlib.h, memory.h (который является ссылкой на string.h), stdio.h. Кроме того функции для работы со строками есть в заголовочном файле math.h.

Рассмотрим минимальный набор функции, позволяющий выполнять любые операции с однобайтовыми строками, реализуемыми с помощью типа char * (указатель на тип char).

Функция strcpy копирует константное значение в другую строку, «стирая» предыдущее значение.

```
char str1[10] = {'h', 'e', 'l', 'l', 'o', 0};
const char str2[] = {'A', 'n', 'n', 0};
printf("%d\n", strlen(str1));
strcpy (&str1[0], str2);
printf("%s\n", str1);
```

Выполнение первой строки приведет к объявлению массива из 10 символов с идентификатором str1, каждый элемент которого имеет тип char. Также в этой строке происходит инициализация массива str1. Первые 5 элементов массива будут заполнены символами hello. Последний инициализируемый элемент массива равен 0.

В соответствии со стандартом языка программирования, все элементы массива, значения которых не указаны при инициализации, становятся равными 0. Соответственно элементы с 5-го по 10-й, имеющие индексы с 4-го по 9-й будут равны. Нумерация символов строк начинается с 0, т. к. указывает на смещение от начала строки (массива), а не на порядковый номер.

Вторая строка исходного текста объявляет строку с идентификатором str2, которая является массивом констант из 4 символов. Последний символ равен 0. Значения символов этой строки изменяться не могут.

В третьей строке производится вызов функции strlen, которая возвращает количество символов в строке до первого нулевого символа. Для str1 длина строки равна пяти символам.

В четвертой строке производится вызов функции strcpy, в которую передаются два массива с помощью аргументов str1 и str2. При этом, первый аргумент передается как указатель на элемент массива str1 с индексом 0, полученный с помощью унарной операции взятия адреса &. Второй аргумент передается с помощью идентификатора строка. В соответствии со стандартом программирования, идентификатор массива интерпретируется как индекс его первого элемента, имеющего индекс 0. В результате первый и второй способ передачи массива по указателю является разнозначным.

После выполнения строки 4 в массиве str1 будут содержаться следующие значения:

```
'A', 'n', 'n', 0, 'o', 0, 0, 0, 0, 0
```

Первые три символа str1 будут заменены символами массива str2. После того как символы для переноса в массиве источника str2 закончатся, функция strcpy

допишет в 4-й символ str1 значение 0. Однако значения элементов str1 с пятого по десятый не изменяться. Так как функции для работы с текстом обрабатывают массив до первого символа 0, буква «о» в пятом элементе массива не будет учитываться при последующей обработке.

Пятая строка содержит вызов оператора printf, который выводит содержимое массива str1, содержащего символы Ann.

Для демонстрации работы функции strcat, дополним приведенные ранее строки исходного текста программы вызовом функции strcat.

```
strcat (str1, "  ");
```

Приведенный вызов содержит имя функции и два аргумента, первый из которой передается с помощью идентификатора str1, второй как константное значение, состоящее из символа пробела. Компилятор формирует из указанной константы массив из двух символов, символа пробела, 0 и передает указатель на этот массив в функцию strcat. Функция strcat находит первый элемент,, равный 0 в массиве str1, начиная с символа 0 переписывает значения в массиве str1, по завершению символов переданных вторым аргументом, записывает после них значение 0. Таким образом, в массиве str1 будут следующие символы и значения:

```
'A', 'n', 'n', '  ', 0, 0, 0, 0, 0, 0
```

Для сравнения строк используется оператор strcmp. Дополним исходный текст следующей строкой:

```
int result = strcmp (str1, str2);
```

В результате выполнения данной строки, переменная с идентификатором result станет равной 32, что равно коду пробела, т. к. массив str1 отличается на один символ пробела, добавленного функцией strcat к str1. Функция strcmp сравнивает посимвольно значения двух массивов и возвращает сумму разностей элементов с соответствующими индексами. Более привычным является использование следующего приема с использованием strcmp:

```
if (strcmp (str1, str2) == 0) {
        ...
        строки равны
        ...
}
else {
        ...
         строки не равны
        ...
}
```

Дополним исходный текст вызовом функции поиска подстроки.

```
char * substring = strstr (str1, "nn");
printf("%s %p %p\n", substring, &str1[1], substring);
```

Вызов этих строк приведет к получению следующего вывода:

```
nn  0x7ffed4dfbd81 0x7ffed4dfbd81
```

Функция strstr, так же как и strcmp, не изменяет

значения в ячейках памяти передаваемых ей аргументов, а возвращает адрес первого вхождения второго аргумента в первый.

Очевидно, последовательность nn входит в str1, начиная со второго элемента, т. е. элемента с индексом 1, поэтому было сделано предположение, что результатом работы функции strstr станет адрес элемента массива str1 с индексом 1. В строке, содержащей вызов printf, указаны спецификаторы типов %s — соответствующий строковым данным и %p — соответствующий адресам переменных в ОЗУ. Следует обратить внимание, переменная substring указана в операторе printf дважды, однако выведены разные значения. Это связано с тем, что первый раз переменная substring была выведена как строка, а второй раз выведен её адрес. Также оператор printf вывел значение адреса элемента массива, полученного выражением &str1[1]; , т. е. адрес элемента массива str1 с индексом 1.

В заголовочном файле определены и другие функции для работы со строками, содержащими однобайтовые символы. Далее приведены их прототипы и назначение.

Функции, аналогичные strcpy и strcat, однако позволяющие ограничивать максимально допустимое для переноса в другую строку количество символов с помощью аргумента n.

*char *strncpy(char * restrict s1, const char * restrict s2, size_t n);*
*char *strncat(char * restrict s1, const char * restrict s2, size_t n);*

Функция, аналогичная strcmp, однако позволяет ограничивать максимально допустимое количество

сравниваемых символов с помощью аргумента n.

*int strncmp(const char *s1, const char *s2, size_t n);*

Функция, аналогичная strcmp, однако учитывающая региональные настройки.

*int strcoll(const char *s1, const char *s2);*

Функция преобразует текст с учетом региональных настроек так, чтобы применение strcmp давало результат, аналогичный вызову strcoll.

*size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);*

Кроме этих функций в разделе 7.24.5 стандарта языка программирования представлены различные функции поиска в тексте.

*void *memchr(const void *s, int c, size_t n);*

Рассматривает строку как массив, состоящий из unsigned char элементов. Осуществляет поиск в первых n байтах по указателю s. Возвращает указатель на первое вхождение символа c в массиве s.

*char *strchr(const char *s, int c);*

Рассматривает строку как массив, состоящий из char элементов. Возвращает указатель на местонахождение первого символа c в массиве s.

*char *strrchr(const char *s, int c);*

Возвращает указатель на местоположение последнего символа c в строке s.

*size_t strspn(const char *s1, const char *s2);*

Вычисляет длину начального сегмента строки s1, состоящего только из символов s2.

*size_t strcspn(const char *s1, const char *s2);*

Вычисляет длину начального сегмента строки s1, не содержащего ни одного символа из s2.

*char *strpbrk(const char *s1, const char *s2);*

Возвращает указатель на первое появление любого из s2 символов в s1.

*char *strtok(char * restrict s1, const char * restrict s2);*

Разделяет строку на последовательность непустых подстрок. При первом вызове указывается строка s1. В s2 указывается набор символов, которые являются разделителями. Для продолжения поиска в последующие вызовы вместо s1 указывается NULL. При последующих вызовах также можно указывать другие разделительные символы. Если найден разделитель, возвращается строка от начала места поиска до разделителя, который заменяется 0. Если встречается несколько разделителей, все они считаются одним разделителем. При достижении конца обрабатываемой строки, функция возвращает NULL.

Пример:

char s1[10] = "*AA1BB12CC11DD*";
char s2[3] = "*123*";
Функция strtok будет возвращать следующие значения
AA
BB
CC
DD

Также в стандарте в файле string.h описаны ряд функций, позволяющих оперировать строками как участками памяти:

*void *memcpy(void * restrict s1, const void * restrict s2, size_t n);*

Копирует из s2 в s1 n байт информации. s2 и s1 не должны перекрываться.

*void *memmove(void *s1, const void *s2, size_t n);*

Копирует из s2 в s1 n байт информации. Если адреса источника и получателя перекрываются, в функции используется вспомогательный буфер.

*int memcmp(const void *s1, const void *s2, size_t n);*

Сравнивает первые n байт в s1 и s2. Возвращет 0, если значения равны. Если массивы не равны, возвращает разницу первых байт сравниваемых массивов.

Для преобразования из однобайтового представления в широкую строку, используются функции, определенные в файле stdlib.h. (7.22.7.3)

*int wctomb(char *s, wchar_t wc);*

Преобразует многобайтовую строку s в строку широких символов pwcs. В pwcs будет записано не более n символов. (7.22.7.1)

*int mblen(const char *s, size_t n);*

Вычисляет количество байт в следующем многобайтовом символе. Функция использует не более n символов строки s. (7.22.7.2)

*int mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);*

Преобразует широкий символ в многобайтовую последовательность, которая записывается в начале строки s. (7.22.8.1)

*size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);*

Преобразовывает мультибайтовую последовательность s в широкий символ pwc. Функция рассматривает не более n символов строки s. (7.22.8.2)

*size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);*

Преобразует широкосимвольную строку в многобайтовую строку. Записывает в s не более n байт.

Почти все функции, применяемые к однобайтовым

строкам, имеют реализации для широких строк. Они объявлены в заголовочном файле wctype.h.

int iswalnum(wint_t wc); - является ли символ числом или буквой.

int iswalpha(wint_t wc); - является ли символ буквой.

int iswblank(wint_t wc); - стандартный символ или региональный.

int iswcntrl(wint_t wc); - другие control символы.

int iswdigit(wint_t wc); - соответствует ли числу.

int iswgraph(wint_t wc); - печатаемый символ и не пробел iswprint = true, iswspace = false.

int iswpunct(wint_t wc); - является ли символом пунктуации.

int iswspace(wint_t wc); - iswalnum, iswgraph, или iswpunct возвращает истина.

int iswupper(wint_t wc); - является ли символ заглавным.

int iswxdigit(wint_t wc); - проверяет, является ли символ шестнадцатиричной цифрой.

int iswctype(wint_t wc, wctype_t desc); - проверяет, имеет ли символ wc свойство desc.

wctype_t wctype(const char *property); - возвращает свойство широкого символа по имени.

wint_t towlower(wint_t wc); - преобразование символа к нижнему регистру.

wint_t towupper(wint_t wc); - преобразование символа к верхнему регистру.

wint_t towctrans(wint_t wc, wctrans_t desc); - транслитерация символа в соответствии с описателем desc.

wctrans_t wctrans(const char *property); - преобразование одного широкого символа в другой.

В заголовочном файле **stddef.h**, описанном в разделе

7.19 стандарта языка программирования, определены типы ptrdiff_t, wchar_t, size_t, макросы NULL, offsetof(). Также в этом файле определены диапазоны для типов wchar_t (WCHAR_MIN и WCHAR_MAX) и wint_t (WINT_MIN и WINT_MAX).

В разделе 6.4.3 стандарта языка программирования указаны способы задания символов по стандарту ISO/IEC 10646 в формате \Unnnnnnnn или \unnnn .

В разделе 7.22.8 стандарта языка программирования описаны функции преобразования мультибайтовых строк в широкие строки и обратно. Примеры применения этих функций

*size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);*

и

*size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);*

доступны на англоязычном сайте cplusplus.com, который содержит примеры почти ко всем языковым конструкциям и функциям.

Для функции mbstowcs() также доступен пример в справочной системе man. Для его отображения необходимо выполнить команду man mbstowcs . Для отображения примера с пояснениями на русском языке необходимо установить пакет **man-pages-ru** и **man-pages-ru-extra** с помощью Synaptic или apt-get.

4.3.3 Пример: инверсия символов

Для обработки строк необязательно использовать готовые библиотечные функции. Приведем пример функции, выполняющей зеркальное отражение однобайтовой строки.

```
1  void word_mirror (char * word) {
2      int i, j;
3      char temp, count_lim = strlen(word) / 2;
4
5      for (i = 0, j = strlen(word)-1; i < count_lim; i++, j--)
6          temp = word[i],
7          word[i] = word[j],
8          word[j] = temp;
9  }
10
```

Функция принимает один аргумент с идентификатором word, имеющий тип указатель на char. Функция ничего не возвращает, а результат её выполнения доступен в точке вызова по указателю на тип char, переданному в функцию.

В строке 2 объявляются две переменные с идентификаторами i и j, имеющие целочисленный тип int.

В строке 3 объявляются две переменные temp и count_lim целочисленного типа char. Переменная count_lim инициализируется результатом выполнения выражения, возвращающего целую часть от деления длины слова word на два, т.е. 5/2=2; 7/3=2. В разделе, посвященном преобразованию типов данных будет описано, почему при делении целого числа на целое, остаток от деления отбрасывается. С помощью такой инициализации решается проблема, связанная с наличием в числе четного и нечетного количества символов. При нечетном количестве символов, средний символ не переставляется.

В строке 5 описан оператор цикла с заданным

количеством итераций. В первом выражении задаются начальные значения двух переменных i и j с использованием операции следования, обозначаемой запятой. Второе выражение определяет условие завершения цикла i < count_limit. В третьем выражении выполняется инкремент счетчика цикла и декремент переменной j. Переменная i увеличивается на единицу при каждом выполнении тела цикла, как бы перемещаясь по первой половине массива. Переменная j перемещается в конец строки и её значение уменьшается на единицу при каждом выполнении тела цикла. Таким образом, она как бы скользит с конца строки на её середину. При достижении обеими переменными середины строки, цикл завершается.

Тело цикла состоит из одного выражения, поэтому не используются фигурные скобки, ограничивающие тело цикла. Однако тело цикла содержит три присвоения, записанные с использованием операции следования. Использование такого приема допустимо только в случае, если среди операций, следующих друг за другом, отсутствуют объявления переменных. Операции со стеком среди операций следования недопустимы.

4.4 Составные типы данных: многомерные массивы

4.4.1 Инициализация многомерных массивов

В основе множества современных математических моделей лежат матричные вычисления. К ним относятся теория распознавания, искусственные нейронные сети, большие данные и другие задачи. В Си для таких задач часто используются многомерные массивы. Объявление многомерных массивов рассматривается в разделе 6.7.6.2 стандарта языка программирования.

Приведем пример инициализации многомерного массива

int m[2][2];

Данное определение можно интерпретировать так, будет создан двухмерный массив, соответствующий матрице 2x2. Выполнение этой строки приводит к выделению памяти на стеке для двухмерного массива. Каждая мерность этого массива состоит из двух элементов целочисленного типа int. Значения в ячейках не определены, поэтому в них будут находиться мусорные значения.

Инициализация многомерных массивов рассмотрена в разделе 6.7.9 стандарта языка программирования. Следуя правилам инициализации многомерных массивов, перепишем приведенное ранее объявление

int m[2][2] = {{11, 22}, {33, 44}};

Допускается инициализация без выделения размерностей фигурными скобками.

int m[2][2] = {11, 22, 33, 44};

В соответствии с приведенной инициализацией, элементы массива будут заполнены следующими константными значениями.

int m[0][0] = 11;
int m[0][1] = 22;
int m[1][0] = 33;
int m[1][1] = 44;

Для определения их расположения в ОЗУ, необходимо вывести адреса ячеек памяти, с которых они начинаются. На используемой автором ПЭВМ были получены следующие значения

int m[0][0] = 0x7ffbbb9a1d70;
int m[0][1] = 0x7ffbbb9a1d74;
int m[1][0] = 0x7ffbbb9a1d78;
int m[1][1] = 0x7ffbbb9a1d7c;

Разность между адресами элементов массива равна 4, т.е. как раз размеру типа int для данной архитектуры микропроцессора. При этом размерности в ОЗУ следуют строго друг за другом без промежутков, сначала расположена размерность с индексом 0, затем с индексом 1.

Следствием такого расположения элементов многомерных массивов является возможность применения адресной арифметики для доступа к элементам многомерного массива.

Например, выражение

**(&m[0][0]+2)*

позволит выполнить следующие действия:

1. выполнена унарная операция взятия адреса элемента массива m с индексами 0, 0;

2. от полученного адреса начального элемента массива m будет выполнено смещение на два элемента размером, соответствующему размеру указателя в сторону увеличения индексов массива. В связи с тем, что в строке массива всего два элемента, указатель будет содержать адрес первого элемента (с индексом 0) элемента следующей строки матрицы;

3. звездочка за скобками означает операцию разыменования, т.е. обращения к значению, содержащемуся по указанному адресу.

Получить адрес элемента можно с помощью следующего выражения

&m[0][0]+2

Ограничения на размерность массива в стандарте языка программирования Си найдено не было. Экспериментально установлено, что компилятор gcc позволяет создавать массивы размерностью более 10, однако в практической работе такие размерности встречаются редко. Рассмотрим несколько наиболее популярных направлений разработки, требующих работы с многомерными пространствами. К ним относятся искусственный нейронные сети состоящие из множества слоев. Например, для распознавания нарисованных символов может использоваться шесть и более слоев. Однако решение этой задачи построено так, что фактически используются двухмерные или одномерные массивы, описывающие каждый слой отдельно. Другим примером являются задачи управления. Как известно, в

природе существует всего четыре вида управления, которые изучает теория управления. Само управление часто сводится к выбору одного из доступных воздействий, в соответствии со сложившейся ситуацией, что также легко реализуемо с помощью одномерного массива. Наибольшая объективно обоснованная размерность массива равна трем. Трехмерные массивы применяются для обработки координат трехмерного пространства.

Также следует учитывать, при увеличении размерности массива значительно увеличивается объем используемой памяти ОЗУ. Двухмерный массив из двух элементов в каждой размерности занимает 16 байт, а десятимерный, в котором каждая размерность равна двум 2096 байт. Создание таких объектов в области стека может привести к выходу за пределы стека и уничтожению программы.

Приведем пример объявления трехмерного массива

char m_3[4][3][2];

Для многомерных массивов допустима частичная инициализация многомерных массивов

char m_3[4][3][2] = {{{ 0 }}, {{ 1, 2 }}, {{ 3 }, { 4, 5 }}};

При частичной инициализации действует правило, в соответствии с которым возможно опускание крайних справа значений, однако недопустим пропуск значений между двумя инициализируемыми значениями. Все не инициализированные значения становятся равны 0.

Для задания значений элементам многомерного массива используются вложенные циклы. Для каждой

размерности необходим свой цикл, поэтому чем больше размерность массива, тем больше вложенных циклов.

```
int array[10][5][3] = {0};
int i = 0;
int j = 0;
for ( ; i < 10; i++)
      for (j = 0 ; j < 5; j++) {
            int k = 0;
            for ( ; k < 3; k++)
                  array[i][j][k] = i+j+k;
      }
```

Следует отметить, что решение задачи вывода всех элементов массива отличается только одной строкой. Вместо присвоения *array[i][j][k] = i+j+k;* необходимо использовать оператор вывода значения *printf("%d\n", array[i][j][k]); .*

Из составных типов можно формировать более сложные составные типы, например, массив структур или структуру, содержащую массивы.

Структура, содержащая массив.

```
struct st {
      int m[10];
      int k[10];
}
```

Массив из 5 структур.

```
struct st [5];
```

Степень вложенности составных типов стандартом языка программирования не ограничена.

Количество элементов многомерного массива, создаваемого на стеке, ограничено размером стека, который может составлять от нескольких килобайт до нескольких мегабайт. В ОС Линукс для архитектуры x86-64 размер стека по умолчанию составляет 8192 Кб. Размер ОЗУ измеряется гигабайтами и чтобы получить к ней доступ необходимо использовать выделяемую память.

Есть три способа выделения динамической памяти для двухмерного массива.

1. Выделить память для одномерного массивов указателей. Затем выделять под каждый одномерный массив память, а его первый адрес заносить в первый элемент базовой размерности, создавая массив указателей на массивы. Преимуществом этого подхода является отсутствие необходимости иметь неразрывный свободный участок памяти, способный вместить все данные. Недостаток состоит в том, что подмассивы будут расположены в памяти не последовательно друг за другом, а в произвольных местах, а значит нельзя будет применять адресную арифметику для работы с элементами массива. Также реализовать такой подход сложнее, чем второй. Его имеет смысл использовать, если требуется захватить как можно больше памяти для хранения данных.

2. Рассчитать размер требуемой памяти и выделить её одним вызовом функции выделения памяти. Адресация к элементам в этом случае производится по указателю с помощью адресной арифметики.

3. Является комбинацией первого и второго. В этом случае создается одномерный массив указателей и следующим этапом выделяется память, достаточная для размещения всех элементов многомерного массива. После этого требуется расставить указатели в массиве

указателей так, чтобы они адресовали ячейки, соответствующие своей размерности.

Подход, связанный с выделением памяти для многомерного массива непрерывным участком, используется чаще всего при условии, что планируется использовать не более 5-10% свободной памяти ОЗУ.

Следует учитывать, операция инициализация массива недоступна для массивов, использующих память из области динамически распределяемой памяти. Также, несмотря на наличие механизмов сбора мусора в современных ОС, после использования многомерного массива, необходимо освободить выделенную под него память.

Первый способ

Рассмотрим более подробно первый способ. Для его демонстрации потребуются следующие определения:

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

#define ELEMENT_QUANTITY 10
#define ARRAY_SIZE
sizeof(int*)*ELEMENT_QUANTITY
#define DIMENSION_SIZE
sizeof(int)*ELEMENT_QUANTITY
```

Далее приведен список функций, используемых в исходном тексте с указанием заголовочных файлов, в которых они описаны.

printf() — stdio.h

malloc() — stdlib.h

free() — stdlib.h
bzero() — string.h

С помощью директив препроцессора определяются три значения. Первое, ELEMENT_QUANTITY, указывает на количество элементов в мерности массива. Второе, *ARRAY_SIZE,* содержит размер в байтах массива указателей на массив. Его размер может отличаться от размера массивов размерностей в случае, если на данной архитектуре размер типа указатель на тип и тип указатель на указатель занимают разный размер в ОЗУ. Третье значение, DIMENSION_SIZE, указывает количество байт занимаемой в памяти размерностью массива array.

Сначала производится объявление переменной с идентификатором array, имеющую тип указателя на указатель на целочисленный тип int. Выделяется память для массива, который будет содержать указатели на другие массивы. В этой же строке производится инициализация переменной array результатом выполнения функции malloc, которая в качестве аргумента принимает размер массива. Запись (int **) обозначает приведение типа результата выполнения функции malloc из типа (void *), который она возвращает после своего завершения, к типу (int **). Более подробно о приведении типов будет рассказано в следующих разделах.

```
int ** array = (int **)malloc(ARRAY_SIZE);
```

Создание второй размерности производится с помощью цикла, приведенного далее

```
int count = 0;
for ( ; count < ELEMENT_QUANTITY; count++)
```

*array[count] = (int *)malloc(DIMENSION_SIZE);*

Сначала производится объявление переменной count целочисленного типа int, которая инициализируется значением 0. Далее следует цикл с заданным количеством итераций, начальное условие которого пропущено, т. к. инициализация счетчика цикла count произведено строкой ранее. Второе выражение предписывает выполнять тело цикла, пока значение счетчика цикла count меньше количества элементов массива ELEMENT_QUANTITY. Третье выражение является инкрементом счетчика цикла count.

Тело цикла состоит из одного выражения, поэтому фигурные скобки, ограничивающие тело цикла, опущены. Телом цикла является вызов оператора выделения памяти в области динамически распределяемой памяти malloc, аргументом которой является размер второй размерности массива array. Функция malloc производит резервирование ячеек памяти ОЗУ, но не изменяет их значений. В этом случае результат выполнения функции malloc приводится к типу (int *) указатель на целочисленный тип, который соответствует одной строке массива.

Далее необходимо обнулить значения полученного двухмерного массива. Для этого также необходимо использовать цикл с заданным количеством итераций.

for (count = 0 ; count < ELEMENT_QUANTITY; count++)
bzero(array[count], DIMENSION_SIZE);

В отличие от предыдущего цикла, его первое выражение задает начальное значение счетчика цикла, равное 0. Телом цикла является вызов функции bzero,

которая обнуляет участок ОЗУ, начиная с указателя переданного ему первым аргументом, размером в байтах, равному второму аргументу.

После создания и обнуления массива обращаться к его элементам можно как элементам многомерного массива, созданного на стеке.

```
array[1][2] = 12;
printf("array[1][2] = %d\n", array[1][2]);
```

После завершения работы с массивом необходимо освободить выделенную под него память. Для этого также используется цикл с заданным количеством итераций.

```
for (count = 0 ; count < ELEMENT_QUANTITY;
count++)
    free(array[count]);
free(array);
```

Телом цикла является вызов функции free(), аргументом которой является указатель на строку массива. Следует обратить внимание, после удаления строк, необходимо удалить сам массив указателей. Для этого в завершении еще раз вызывается функция free() с аргументом array.

Второй способ

Реализация второго способа требует использования следующих функций:

printf() — stdio.h
calloc() — stdlib.h
free() — stdlib.h

При реализации этого способа используется функция calloc(), которая не только резервирует память в ОЗУ, но и обнуляет её. В связи с этим использование специальной функции для заполнения ячеек ОЗУ нулями не требуется.

В примере используется значение ELEMENT_QUANTITY, определенно директивой предпроцессора и равной 10. Соответственно будет создан массив размером 10x10 элементов.

```
#define ELEMENT_QUANTITY 10
```

Для получения доступа к элементам массива используется переменная array, которая имеет тип указателя на целочисленное значение типа int. Значения такого типа используются как указатели на первый элемент одномерного массива.

```
int * array =
calloc(ELEMENT_QUANTITY*ELEMENT_QUANTITY,
                    sizeof(int));
```

Для обращения к элементам таких массивов используется адресная арифметика. Для повышения читаемости исходного текста используются вспомогательные переменные column_size, row_size, column, row и array_ptr. Их назначение:

column_size — количество элементов в столбце;

row_size — количество элементов в строке;

column — текущий номер столбца;

row — текущий номер строки;

array_ptr — указатель на элемент массива.

В приведенном далее исходном тексте в первой строке объявляется четыре переменных целочисленного типа int. Следующая строка задает значение

ELEMENT_QUANTITY для column_size и row_size, причем присвоение производится справа налево. Таким же образом задается значение 1 для column и row.

Для обращения к элементу массива объявляется переменная array_ptr, которая инициализируется результатом выполнения суммирования адреса array, указывающего на первый элемент массива, и результатом выражения column*row_size+row. Выражение указывает на порядковый номер элемента так, как будто все последующие строки расположены в памяти последовательно за первой с индексом 0. По правилам языка Си, если к адресу массива прибавляется число, то адрес увеличивается на размер элемента массива, умноженный на прибавляемое число. Таким образом производится переход на соответствующий элемент массива. В результате array_ptr будет указывать на элемент с индексами [1][1].

```
int column_size, row_size, column, row;
column_size = row_size = ELEMENT_QUANTITY;
column = row = 1;
int * array_ptr = array+(column*row_size+row);
*array_ptr = 1;
printf("array[1][1] = %d\n",
array[column*row_size+row]);
```

Для обращения к элементу массива выполняется операция разыменования, обозначаемая символом звездочка, и присваивается значение 1.

Для подтверждения правильности задания значения для нужного элемента массива используется функция printf(), выводящая значение соответствующего элемента. Второй аргумент функции printf() также использует правила адресной арифметики, однако её

синтаксис отличается. Данное обращение больше похоже на указание индекса элемента в одномерном массиве.

Освобождение памяти производится с помощью однократного вызова функции free().

free (array);

Третий способ

Для реализации третьего способа необходимы те же функции и определения как для второго способа. При этом требуется выделение памяти для двух массивов, массива указателей array и массива значений array_ptr. Массивы отличаются не только количеством элементов, но и типом элементов. Количество элементов массива указателей равно количеству столбцов в матрице, а количество элементов в массиве значений равно произведению количества элементов в столбце на количество элементов в строке.

```
int ** array = calloc(ELEMENT_QUANTITY,
sizeof(int*));
int * array_value =
calloc(ELEMENT_QUANTITY*ELEMENT_QUANTITY,
        sizeof(int));
```

Для данного способа характерен дополнительный этап подготовки массивов, связанный с получением адресов начала строк массива значений. После выполнения приведенного ниже цикла значениями элементов массива array станут адреса элементов массива array_value.

```
int count = 0;
```

```
    for ( ; count < ELEMENT_QUANTITY; count++)
        array[count] =
&array_value[count*ELEMENT_QUANTITY];
```

После этого обращаться к элементу массива можно как к элементу многомерного массива.

```
array[2][2] = 18;
printf("array[2][2] = %d\n", array[2][2]);
```

Для освобождения выделенной под массивы памяти необходимо два вызова функции free(). Если функции free вызываются по-очереди, порядок их вызова не так важен. Однако правильнее сначала удалить массив указателей, затем массив значений, что позволит исключить обращение по адресу освобожденной памяти.

```
free (array);
free (array_value);
```

Рассмотренные способы выделения памяти для двухмерных массивов применимы к массивам любой мерности, однако с добавлением каждой мерности, повышается сложность алгоритмов и понижается читаемость исходных текстов программы.

Приведем пример обработки трехмерного массива с помощью первого способа. В данном примере для выделения памяти ОЗУ будет использоваться фукция calloc, поэтому обнуление выделенной памяти не потребуется.

printf() — stdio.h
calloc() — stdlib.h
free() — stdlib.h
Количество элементов в каждой размерности

определяется с помощью директив предпроцессора.

```
#define DIMENTION1 3
#define DIMENTION2 4
#define DIMENTION3 5
```

Для выделения памяти потребуется несколько предварительных действий. Сначала необходимо объявить переменную array, которая будет являться указателем на массив. Формально её типом является указатель, на указатель, на указатель на целочисленный тип int. В ОЗУ она хранится как указатель на тип int. До создания массива объявляются и инициализируются значением 0 счетчики цикла для каждой размерности i, j, h.

```
int ***array;
int i, j, h;
i = j = h = 0;
```

Создание массива отличается от создания двухмерного массива дополнительным циклом for, который создает еще одну размерность со значениями типа указателей, хранящихся во вспомогательных массивах. Указание в скобках операций явного приведения типа перед вызовами функции calloc() призваны подчеркнуть, какие именно типы используются для хранения указателей. Внешний цикл for добавляет в каждый элемент массива array адрес начальной ячейки массива размером DIMENTION2. Вновь создаваемым массивам не присваивается отдельное имя, поэтому к ним можно будет обращаться только через адрес в ячейке массива array. Внутренний цикл for добавляет в каждую ячейку всех массивов, создаваемых внешним

циклом for адреса начальных ячеек массивов, которые будут содержать обрабатываемые данные.

```
array = (int ***)calloc(DIMENTION1, sizeof(int**));
for ( ; i < DIMENTION1; i++) {
   array[i] = (int **)calloc(DIMENTION2, sizeof(int*));
   for (j = 0; j < DIMENTION2; j++)
      array[i][j] = (int *)calloc(DIMENTION3,
sizeof(int));
}
```

Вспомогательными являются массив array (обращение к элементам array[x]) и все массивы, к элементам которых обращение производится с помощью конструкций array[x][y], где x от 0 до значения константы DIMENTION1, а y от 0 до DIMENTION2. Сами значения хранятся в массивах, к которым обращение производится с помощью конструкций вида array[x][y][z], где x изменяется от 0 до DIMENTION1, y от 0 до DIMENTION2, а z от 0 до DIMENTION3.

```
int count = 0;
for (i = 0; i < DIMENTION1; i++) {
   for (j = 0; j < DIMENTION2; j++) {
      for (h = 0 ; h < DIMENTION3; h++) {
         array[i][j][h] = count++;
         printf("array[%d][%d][%d] = %d ", i, j, h,
array[i][j][h]);
      }
      puts("");
   }
   puts("");
}
```

Для освобождения памяти, выделенной для трехмерного массива, необходимо выполнить следующие операции. Внутренний цикл for удаляет все массивы с данными, обращение к которым производится с помощью операции array[x][y][z]. Внешний цикл for освобождает память, выделенную под вспомогательные массивы с указателями, к которым обращение производится с помощью операций array[x][y]. Последний вызов функции free() освобождает память, выделенную под вспомогательный массив с указателями array.

```
for (i = 0; i < DIMENTION1; i++) {
   for (j = 0; j < DIMENTION2; j++)
      free(array[i][j]);
   free(array[i]);
}
free(array);
```

Все три способа выделения памяти под многомерные массивы позволяют создавать массивы любой размерности. При увеличении размерности массива происходит возрастание сложности алгоритмов выделения и освобождения памяти для хранения его значений.

Приведенные примеры демонстрируют некоторое неудобство при работе с многомерными массивами, память для которых выделяется в области динамической памяти. При практическом программировании редко встречаются задачи, требующие использования массивов размерностью более 3, а использование массивов высокой размерности часто является показателем алгоритмических ошибок и ошибок при проектировании.

4.4.2 Операции с многомерными массивами

Современные математические задачи обычно используют пространства стабильной размерности. Если в задаче используется преобразование с разными размерностями, выбирается массив с максимально возможной размерностью или два массива с разной размерностью.

При решении задачи сжатии данных до меньших размерностей, т. е. задач понижения размерности или свертки, неиспользованная память ОЗУ остается выделенной до завершения расчета. Кроме того такие задачи обычно решаются отдельно от основной задачи и используют сразу несколько массивов разной размерности.

Как уже было показано в предыдущем пункте, увеличение размерности массива приводит к увеличению сложности алгоритмов, описывающих операции с ним, а также увеличению используемой памяти. Особое внимание следует уделить рекурсивным функциям, которые многократно вызывают сами себя. В случае их использования не рекомендуется использовать память для многомерного массива в области стека.

Основные и наиболее часто встречающиеся операции с многомерными массивами:

1) выделение и освобождение выделенной памяти под массив;

2) обнуление массива;

3) доступ к произвольному элементу массива;

4) последовательный доступ ко всем элементам массива;

5) вывод значений массива в устройство вывода.

В предыдущем разделе предложено три способа выделения памяти для многомерных массивов. Для

первого способа обращение к элементам возможно только с помощью квадратных скобок. Второй способ допускает использование только адресной арифметики. Третий дает возможность использовать как квадратные скобки, так и адресную арифметику.

Операции пунктов 1, 2 и 3 рассмотрены в предыдущем разделе.

Последовательный доступ к каждому элементу массива осуществляется с помощью вложенных циклов, количество которых равно количеству мерностей массива. Пример последовательного доступа ко всем элементам массива предыдущего раздела использует трехмерный массив. В этом примере операция доступа к элементам массива дополнена функцией вывода значения элемента массива в стандартный поток вывода printf(), а значит реализует вывод значений всех элементов массива.

Рассмотрим случай, при котором требуется инициализировать массив значением, отличным от 0. Такая необходимость возникает, когда требуется отслеживать, использован элемент массива или нет. Для этого при разработке алгоритма выбирается «неиспользуемое» значение, которое и означает, что ячейка не используется. Проще всего выбрать неиспользуемым значением недопустимое с точки зрения предметной области решаемой задачи. Например, столкновение физических тел подразумевает наличие как минимум двух физических объектов, поэтому неиспользуемым значением может быть как 0, так и 1. В случае, когда такие недопустимые значения отсутствуют, «неиспользуемым» выбирается 0, минимальное или максимальное значение, допустимое для данного типа данных языка программирования.

Пример побайтового задания значения 255 всем

ячейкам памяти с помощью функции memset() приведен в пункте 4.2.3. Инициировать ячейки памяти также можно значениями типа wchar_t с помощью функции wmemset().

Кроме задания «неиспользуемого» значения, возможно объявление составного типа struct для выставления флагов неиспользуемых элементов.

```
struct value {
    int data;
    bool enabled;
}
```

Стоит обратить внимание на тип bool, для применения которого необходимо использовать директиву препроцессора #include "stdbool.h" (7.18 стандарта языка программирования). Данный тип реализован с помощью директив препроцессора, которыми переопределяется тип _Bool и позволяет использовать значение true и false в тексте программы. В связи с этим, тип bool также, как и тип _Bool , занимает в памяти 1 байт, а не бит. Современные архитектуры микропроцессоров не позволяют адресовать один бит, что можно проверить с помощью выражения

```
printf("sizeof(_Bool) = %ld\n", sizeof(bool));
```

Допустимо объявление многомерного массива, состоящего из структур типа value.

```
struct value array[10][2];
```

Работа с массивом структур выполняется также, как и с обычным массивом, за исключением необходимости

уточнять, к какому именно элементу структуры производится обращение. Например:

array[count1][count2].enabled ? array[count1] [count2].data : 0;

В указанном примере в случае, когда элемент структуры enabled равен 1, возвращается значение элемента структуры data, в противном случае возвращается значение 0. Данный пример наглядно демонстрирует потерю читаемости текста программы при увеличении размерности массива.

Кроме типа указатель на указатель, int ** array, в котором количество звездочек указывает на размерность массива, существует еще один способ объявления указателей на массив. Например,

int ([4])[3];*

объявляет массив из 4-х указателей на массивы, содержащие по 3 элемента.

Стандарт языка программирования допускает использование сечений многомерных массивов. Ограничение состоит в невозможности использования промежуточных размерностей для передачи сечения. Например, для int array[5][4] :

- array[2] — допустимо, указывает на подмассив из 4 элементов, который следует третьим от начала двухмерного массива, т. е. соответствует третьей строке;

- array[][3] — недопустимо, нельзя опускать размерность после указанной, т. е. нельзя передать указатель на столбец двухмерного массива.

Сечения массивов используются в основном в многопоточном программировании, для понижения

размерности данных обычно используется отдельный массив необходимой размерности.

Начинающие программисты часто задаются вопросом, возможно ли написание универсальной функции, осуществляющей ввод или вывод значений любой размерности. Да, при передаче размерностей массивов отдельными аргументами это возможно, но в практическом программировании лучше иметь отдельные функции для вывода значений массивов различной размерности. Кроме того, функции поиска, обхода всех значений также лучше писать в каждом случае отдельно, т. к. не существует единого оптимального алгоритма работы с многомерными массивами для всех вариантов расположения их элементов в ОЗУ. При использовании объемов информации, сопоставимых с физически доступным ОЗУ, требуется разработка специальных алгоритмов для выделения и освобождения памяти ОЗУ.

При передачи многомерных массивов в функцию могут использоваться следующие приемы:

- функция принимает указатель на массив, зная его размерность

*int print_array_5_3_5 (int *** array);*

- функция принимает указатель на первый элемент трехмерного массива, что определяется типом аргумента, передаваемого в функцию. В идентификаторе функции указаны длины этих размерностей. Соответственно функция сможет печатать только массивы типа

int array[5][3][5];

- функция принимает указатель на массив с объявленными мерностями

print_array(int array[5][10]);

приведенная функция также ожидает только многомерные массивы указанной мерности;

- функция принимает указатель и длины всех размерностей в качестве аргументов;

*int print_array (int *** array, int dimention1, int dimention2, int dimention3);*

такая функция должна уметь печатать значения всех трехмерных массивов, независимо от длин его размерностей.

Раздел содержит пояснения практически без приведения исходных текстов программы по нескольким причинам. Исходный текст обработки многомерных массивов громоздкий и занимал бы слишком много страниц. Он частично приведен в предыдущих пунктах. Информация по этой теме имеется на общедоступных сетевых ресурсах, и обычно не вызывает затруднений. Кроме того, умение писать исходный текст программы по устному описанию является обязательным навыком для студентов старших курсов.

4.5 Составные типы данных: битовые поля

Битовые поля являются вспомогательным средством, реализующим доступ к определенному биту, используя побитовые операции с битовыми масками, но использующие синтаксис, применяемый к составным типам данных. В практических задачах, единственной областью в которой они используются регулярно — разложение байт, полученных с помощью аппаратных интерфейсов, на битовые флаги.

В стандарте языка программирования отсутствует специальный раздел, посвященный этому средству. Основные синтаксические конструкции для работы с этим средством полностью совпадают с применением структур, объединений и перечислений.

Битовые поля желательно использовать только в модулях, производящих аппаратнозависимую обработку данных. Расположение бит битового поля зависит от архитектуры процессора, а вопрос выравнивания битовых полей в структурах может зависеть от множества факторов. Также применительно к битовым полям не имеют смысла операции получения адреса, т. к. минимально адресуемым адресным пространством на аппаратном уровне является байт.

На рисунке 1 приведен исходный текст программы, демонстрирующей основные приемы работы с битовыми полями. В приведенном примере битовые поля объявлены в структуре bitfield, которая объявлена в строках 4-11. Структура содержит 4 однобитовых поля в строках 6-9 и одно четырехбитовое в строке 10, которые совместно формируют один байт.

В строках 12-15 содержится объявление объединения bfvalue, состоящее из поля бит bitfield и переменной с идентификатором value типа unsigned char.

В строке 17 объявляется переменная типа union bfvalue с идентификатором bfv.

В строке 18 переменная bfv инициализируется значением 0, с использованием поля объединения value.

В строке 20 выводиться значение value объединения bfv сразу после инициализации объединения.

В строке 21 элементу битового поля b1 присваивается значение 1.

В строках 22, 24 выводится значение поля b1 и значение value.

В строке 23 однобитовое значение b1 увеличивается на 2.

В строке 26 элементу битового поля b5 размером 4 бита присваивается значение 1.

В строках 27, 29 выводится значение поля b5 и значение value.

В строке 28 элементу битового поля b5 размером 4 бита присваивается значение 15.

Строка 30 содержит оператор return, который возвращает значение 0 в точку вызова, т. к. оператор находится в функции main, его использование приводит к завершению программы и освобождению всех ресурсов выделенных её ОС.

Результат выполнения исполняемого файла, полученного из приведенного текста программы, представлен на рисунке 2.

В результаты выполнения строки 20 исходного текста программы на экран выведено значение элемента value = 0.

Вторая и третья строка вывода получена в результате выполнения строки 22 исходного текста программы. Значение поля b1 стало равно 1 и значение value также изменилось на 1. Это связано с архитектурой микропроцессора x86-64, которая располагает данные по

правилу big ending.

01.c

```c
#include <stdio.h>

int main () {
    // Битовое поле содержит однобайтовые и многобайтовые элементы
    struct bitfield {
        unsigned b1: 1;
        unsigned b2: 1;
        unsigned b3: 1;
        unsigned b4: 1;
        unsigned b5: 4;
    };
    union bfvalue {
        struct bitfield bf;
        unsigned char value;
    };
    // Инициализация объединения
    union bfvalue bfv;
    bfv.value = 0;

    printf("value = %d\n",bfv.value);
    bfv.bf.b1 = 1;
    printf("b1 = 1, %d\nvalue = %d\n", bfv.bf.b1, bfv.value);
    bfv.bf.b1+=2;
    printf("b1 += 2, %d\nvalue = %d\n", bfv.bf.b1, bfv.value);

    bfv.bf.b5 = 1;
    printf("b5 = 1, %d\nvalue = %d\n",bfv.bf.b5, bfv.value);
    bfv.bf.b5 = 15;
    printf("b5 = 15, %d\nvalue = %d\n",bfv.bf.b5, bfv.value);
    return 0;
}

```

Рисунок 1 — Исходный текст программы, демонстрирующей базовые операции с битовыми полями

Четвертая и пятая строка вывода сформирована 24 строкой исходного текста программы. Следует обратить внимание на то, что в 23 строке исходного текста однобитовое значение было увеличено на 2. В случае обычного сложения 1+2=3, а как известно для записи числа 3 необходимо минимум два бита. Однако, в данном

случае, операция увеличения на 2 воспринимается не как сложение, а как двухкратная инверсия поля b1. Соответственно, если бы значение увеличивалось на 3, то b1 было бы рано 0.

Шестая и седьмая строка вывода получены 27 строкой исходного текста. Полю b5 присваивается значение 1. В свою очередь, value становиться равно 17, т. к. 2^5=16 плюс значение b1 равно 1. В результате получается 17.

Две последние строки вывода сформированы строкой 29 исходного текста программы. После присвоения полю b5 значения 15, все биты переменной value кроме 2,3 и 4 стали равны 1, что соответствует значению 241.

```
value = 0
b1 = 1, 1
value = 1
b1 += 2, 1
value = 1
b5 = 1, 1
value = 17
b5 = 15, 15
value = 241
```

Рисунок 2 — Вывод ЭТ с результатами работы программы

Рассмотрим другой пример объявления типа структуры с использованием поля бит.

```
#pragma pack(push,1)
struct bf2 {
    int b1 : 1;
    int b2 : 1;
    : 3;
    int k;
```

```
};
#pragma pop()
```

При её объявлении использовались следующие приемы. С помощью директивы #pragma выполнена упаковка структуры, т. е. данные должны следовать друг за другом, без пропусков байт в ОЗУ. Третье битовое поле не имеет имени, а значит обратиться к этим битам с помощью этой структуры не получится. Переменная k типа int будет расположена в байтах, следующих за пределами байт, использованных для битовых полей. В связи с тем, что количество байт, занимаемых типом int на разных архитектурах может быть разным, использовать такую структуру как универсальное решение нельзя даже в упакованном виде.

В вопросах, касающихся выравнивания структур, включающих в себя битовые поля, следует руководствоваться правилами общими для всех других случаев выравнивания структур, описанных в пункте 4.2.3.

4.6. Составные типы данных: перечисления

Перечисления — это языковая конструкция, реализующая подмену целочисленных констант типа int на комбинацию символов в исходных текстах программ. Как и битовые маски, перечисления предназначены для повышения читаемости исходного текста программ. Они описаны в разделе 6.7.2.2 стандарта языка программирования Си. В указанном разделе стандарта языка программирования приведен озорной пример с использованием наименований алкогольной продукции, однако в этом разделе книги будут использована другая предметная область.

Рассмотрим пример с использованием перечислений.

01.c

```c
#include <stdio.h>

int main () {
    enum rgb { red, green, blue };
    printf("sizeof %ld\nred %d\ngreen %d\nblue %d\n",
            sizeof (enum rgb), red, green, blue);
    enum color { Red, Orange = 10, Yellow, Green = 5 , Blue };
    enum color c;
    printf("sizeof %ld\nRed %d\nOrange %d\nYellow %d\nGreen %d\nBlue
    %d\n",
            sizeof (c), Red, Orange, Yellow, Green, Blue);
    int color_example = Orange;
    printf( "color_example %d\n", color_example);
    if (red == Red)
        printf("red == Red, %d == %d\n", red, Red);
    if (green != Green)
        printf("green != Green, %d != %d\n",
               green, Green);
    return 0;
}
```

Рисунок 1 — Исходный текст программы, демонстрирующей приемы работы с перечислениями

В приведенном примере опишем работу строк, касающихся непосредственно работы с перечислениями.

Строка 4. Объявляется перечисление enum rgb с элементами red, green, blue.

Строки 5-6. Вызывается функция форматированного вывода printf, которая, в соответствии с шаблоном вывода отобразит размер перечисления enum rgb, константные значения, соответствующие red, green, blue.

В строке 7 объявляется перечисление color. Перечисление содержит элементы Red, Orange, Yellow, Green, Blue, причем элементу Orange присвоено константное значение 10, а элементу Green значение 5.

В строке 8 объявляется значение c, которое в дальнейшем нельзя использовать как переменную, например конструкций c = Red, не будет иметь для компилятора смысла. Идентификатор c допустимо использовать вместо конструкции enum color, например в операторе sizeof.

Строки 9-10 содержат вызов функции printf, которая выводит размер по идентификатору c и константные значения для всех элементов enum color. Следует обратить внимание: строка 9 располагается на двух строках, так как в редакторе Geany, используемом для работы с исходным текстом программы, включен режим переноса строк.

В строке 11 объявляется переменная с идентификатором color_example типа int, которая инициализируется значением элемента Orange.

Строка 12 содержит printf, выводящий в стандартный поток вывода значение переменной orange_color.

В строке 13 выполняется сравнение двух значений, объявленных в разных перечислениях. В случае, если эти значения равны, выполняется вызов функции printf в строке 14, содержащей в своем шаблоне вывода

сообщение о равенстве значений и отображающей значения элементов red и Red.

В строке 15 с помощью оператора условного перехода if выполняется сравнение значений разных объединений green и Green и в случае их неравенства выполняется вызов функции printf в строке 16. В строке 17 записаны второй и третий аргумент функции printf предыдущей строки.

Вывод программы, полученной из исходного текста, представленного на рисунке 1, показан на рисунке 2.

```
sizeof 4
red 0
green 1
blue 2
sizeof 4
Red 0
Orange 10
Yellow 11
Green 5
Blue 6
color_example 10
red == Red, 0 == 0
green != Green, 1 != 5
```

Рисунок 2 — Результат выполнения программы

Строки вывода 1, 2, 3, 4 сформированы строками 5, 6 исходного текста программы. Размер перечисления enum rgb определен равным 4, что соответствует размеру типа int, как и указано в пункте 2 раздела 6.7.2.2 стандарта языка программирования Си. Данные строки демонстрируют, если константные значения элементам перечисления не задаются, то они нумеруются начиная с 0, каждый следующий элемент на 1 больше предыдущего.

Строки вывода 5, 6, 7, 8, 9, 10 сформированы

строками 9 и 10 исходного текста программы. Они демонстрируют, что в случае задания константных значений элементам перечисления, последующие не инициированные значения имеют значение на единицу больше предыдущего. Таким образом, первое значение Red равно 0, т. к. оно не инициировалось. Значение Orange инициировано в строке 7 исходного текста программы и равно 10. Следующее значение Yellow не инициировано. Его значение задано как на единицу большее Orange, т. е. 11. Следующее значение Green инициировано и равно 5. Значение Blue не инициализировано и следует за Green = 5, поэтому равно 6.

Третья снизу строка вывода отображает значение color_example, которое в строке 11 исходного текста программы инициируется значением Orange. Как видно из предыдущего вывода, Orange равно 10.

Вторая снизу строка вывода сформирована строкой 14 исходного текста программы. Вывод сообщает, что значения элементов red и Red из разных перечислений равны. Они не были инициированы и с них начинаются оба перечисления, поэтому они равны 0.

Последняя строка вывода сообщает о том, что значения элементов green и Green разных перечислений не равны green =1, а Green равно 5.

С помощью последних двух строк вывода продемонстрирована причина ошибок, возникающих при использовании нескольких объединений для одних и тех же типов или сущностей в одном модуле или программе. Таким ошибкам подвержены программы, разные части которых написаны разными разработчиками, а объединением их в один продукт проводится третьим разработчиком.

Вместо перечисления enum color возможно

использование нескольких директив предпроцессора #define, например так:

```
#define red      1
#define orange 2
#define yellow  3
#define green   4
#define blue    5
```

После этого становятся допустимы выражения типа:

```
int color = red;
```

В идеальном случае в одной программе не должно быть нескольких перечислений, описывающих одни и те же понятия. Необходимо заранее определиться, будет перечисление использоваться для замены константных числовых значений их обозначениями или также необходимо задавать и интерпретировать каким-либо образом сами константные значения. Второй случай усложняет читаемость исходных текстов программ, однако позволяет разработать правила применения операций с элементами перечисления.

5. Язык программирования Си. Вспомогательные средства и приемы работы

5.1. Переменные. Указатели. Объявления

В практическом программировании из целочисленных типов выбираются те, что соответствуют машинному слову на целевой машине. Обычно именно такие типы обрабатываются быстрее, а типы меньшего размера эмулируются за счет типов, соответствующих минимальной разрядности микропроцессора. При использовании составных типов ситуация меняется и необходимо использовать принцип выбора типа данных, занимающего минимальный размер в ОЗУ.

При выборе типа с плавающей запятой обычно делают выбор в пользу типа double, т. к. он используется в большем количестве прототипов библиотечный функций, а значит будет проще писать программы. Тип float предпочитают не использовать, т. к. он не поддерживается на некоторых специализированных платформах несмотря на его наличие в стандарте языка программирования.

Инициализация переменных с плавающей запятой требует задания значения как целой, так и дробной части

```
double dbl = 0.0;
```

Кроме того, при использовании типов с плавающей запятой нельзя выполнять непосредственное сравнение на равенство

```
double dbl1 = 0.0;
double dbl2 = 0.0;
...
```

if (dbl1 == dbl2) // Условие практически никогда не будет выполнено

Обычно для сравнения переменных с плавающей запятой, их предварительно округляют с требуемой точностью. Пример округления до 100-й доли

```
double dbl1 = 12345.123456;
double dbl2 = 12345.128456;
int temp = (int)((double)dbl1*(double)100.0);
double dbl1_temp = (double) temp/100;
temp = (int)((double)dbl2*(double)100.0);
double dbl2_temp = (double) temp/100;
if (dbl1_temp == dbl2_temp)
    puts("Значения равны");
```

Приведенный пример фрагмента исходного текста программы выведет ответ «Значения равны», несмотря на различия в тысячных долях значений переменных dbl1 и dbl2. Пояснение его работы приведено в разделе 5.2 этой книги.

Для значений типов с плавающей запятой применимы операции < и >, как и для целочисленных типов.

Отдельным классом типов данных являются указатели. Проводя аналогию, их можно сравнить с номерами мобильных телефонов. Знать номер мобильного телефона, не значит знать то, что знает его владелец, однако используя этот номер, можно связаться с владельцем и узнать требуемые данные.

Указатели позволяют передавать данные, не перемещая их между устройствами и ячейками памяти. Вместо данных передается указатель на эти данные и таким образом получается выигрыш в

производительности и затрачиваемых ресурсах. Соответственно, чем больше данных требуется передать, тем больше выигрыш при передаче указателя.

Не все типы данных хранятся в ОЗУ как последовательность байт, не требующая специальной интерпретации, поэтому размеры указателей на разные типы могут различаться. Кроме того размеры типов указателей могут измениться при переходе на другую платформу.

Для определения размера ОЗУ в байтах, необходимого для хранения указателя на тип, можно использовать выражение

printf("sizeof char = %ld bytes\n", sizeof (char *));*

Все типы данных, включая составные типы и типы функций, могут использоваться для составления сложного описателя типа данных.

При инициализации сложных описателей типов действуют правила приоритета. Квадратные и круглые скобки интерпретируются раньше звездочки, но имеют равный приоритет между собой и раскрываются слева направо. Спецификатор типа является последним элементом подвергающимся интерпретации.

Для интерпретации сложных описаний используется правило «изнутри наружу», состоящее из четырех шагов.

1. Интерпретация начинается с идентификатора в центре, затем необходимо посмотреть направо, есть ли там квадратные или круглые скобки.

2. Если на предыдущем этапе обнаружены скобки, необходимо интепретировать эту часть описателя и проверить слева наличие символа звездочка.

3. Если на любой стадии справа встречается закрывающая круглая скобка, то вначале правила

применяются внутри круглых скобок, затем выйти за них и начать с правила 1, считая идентификатором результат интерпретации в скобках.

4. Интерпретировать спецификатор типа.

Примеры:

*float (* funct_ptr) (char value1, char value2);*
4 2 1 3

Объявляется функция funct_ptr (1) как указатель (2) на функцию, принимающую два аргумента типа char (3), возвращающую значение типа float.

*char * (* (* func_ptr) ()) [5];*
7 6 4 2 1 3 5

Объявляется функция func_ptr (1) как указатель (2) на функцию (3), возвращающую указатель (4) на массив (5) из пяти элементов, которые являются указателями (6) на значения типа char.

*int * (* array_func_ptr [5]) ();*
6 5 3 1 2 4

Объявляется переменная array_func_ptr (1), как массив из пяти (2) указателей (3) на функции (4), возвращающие указатели (5) на значения типа int (6).

Описатели такого уровня вложенности часто говорят о попытке использовать объектно-ориентированный подход в функциональном программировании или при возникновении других проблем проектирования. Объективные причины использования сложных типов

функций — это создание и работа с динамически подключаемыми библиотеками, а также объявление прототипов функций.

Для повышения читаемости исходного текста программ, использующих сложные описатели, возможно применение объявления типов. Этот прием описан в стандарте языка программирования Си в разделе 6.7.8 Объявление типов. Объявление пользовательских типов производится как с помощью тегов, так и с помощью ключевого слова typedef.

Примеры:

Объявляется тип указатель на функцию с идентификатором activation. Её входными аргументами являются два массива неопределенной длины a и b. Функция возвращает значение типа int.

int (function)(int a[], int b[]);*

С помощью ключевого слова typedef возможно определение типа функции, указанной ранее

typedef int (activation)(int a[], int b[]);*

Именем нового типа данных является activation. После этого возможно объявление указателя на функцию с идентификатором accumulator.

activation accumulator;

Представленное объявление функции не содержит в себе символа звездочка, который указывает на то, что объявляется указатель. В случае, когда объявление

находится в другом файле и поиск его объявления затруднителен, читаемость исходного текста значительно понижается.

При работе с библиотеками необходимо создавать указатели для вызываемых функций. Создание указателя на функцию синтаксически мало отличается от создания указателя для переменной. Например, для функции

int func ();

указатель получается следующим образом

*int (*func_ptr)() = &func;*

После этого с указателем на функцию можно работать как с указателем на переменную.

В стандарте языка программирования Си в разделе 7.19 «Дополнительные объявления» приведены объявления, которые часто встречаются в практической работе. Описания этих объявлений находятся в файле stddef.h.

К ним относятся следующие типы:

- ptrdiff_t - используется знаковый целый тип для хранения результата вычитания между двумя адресами;
- size_t - используется беззнаковый тип для хранения результата выполнения операции sizeof;
- max_align_t — используется как тип базовых объявлений, описанных в пункте 6.2.8 стандарта языка программирования Си.

В файле stddef.h также приведены допустимые диапазоны для этих типов, например

PTRDIFF_MIN -65535
PTRDIFF_MAX +65535

5.2. Преобразование типов данных

В языке программирования Си присутствует как явное преобразование типов данных, так и не явное. При явном преобразовании указывается тип, к которому выполняется приведение значения. Неявное преобразование осуществляется по определенным правилам, оговоренным в стандарте. В процессе разработки рекомендуется не использовать неявное приведение типов. Во время компиляции, компилятор выдает предупреждение на многие виды неявного преобразования типов.

К основным правилам и ограничениям неявного преобразования данных разных типов можно отнести:

- во время вывода значения типа char в поток вывода он будет интерпретирован как символ, а не как число;
- при присвоении значения типа, занимающего в ОЗУ больше ячеек памяти, типу, занимающему меньше ячеек памяти, возможна потеря данных;
- при присвоении значения типа с плавающей запятой, целочисленному типу, дробная часть отбрасывается;
- при присвоении значения беззнакового типа знаковому возможна инверсия знака числа.

Неучтенное изменение значения при приведении типов не считается ошибкой с точки зрения языка программирования Си. Ответственность за них полностью ложится на плечи программиста.

Приведем пример исходного текста программы (Рисунок 1), который демонстрирует отображение участка ОЗУ одного целочисленного типа, с помощью других типов данных.

В строке 4 объявляется объединение, включающее в себя переменные типов char, int и unsigned int, причем в

одних и тех же ячейках ОЗУ. Для объединения будет выделен размер памяти, равный наибольшему типу, т. е. int или unsigned int. Эти типы занимают одинаковый объем ОЗУ. Объединение не использует тег, поэтому использовать его можно будет только один раз. При объявлении объединения создается его экземпляр с идентификатором diff_type.

01.c

```
1  #include <stdio.h>
2
3  int main () {
4      union { char ch; int si; unsigned int usi; } diff_type;
5
6      diff_type.ch = 'A';
7      printf("%c %d %d %d\n", diff_type.ch, diff_type.ch,
8                              diff_type.si, diff_type.usi);
9      diff_type.usi = 0;
10     diff_type.ch = 'A';
11     printf("%c %d %d %d\n", diff_type.ch, diff_type.ch,
12                             diff_type.si, diff_type.usi);
13     diff_type.usi = -65;
14     printf("%c %d %d %d\n", diff_type.ch, diff_type.ch,
15                             diff_type.si, diff_type.usi);
16     diff_type.usi = -65000000;
17     printf("%c %d %d %u\n", diff_type.ch, diff_type.ch,
18                             diff_type.si, diff_type.usi);
19     return 0;
20 }
21
```

Рисунок 1 — Исходный текст программы с целыми типами данных.

В строке 6 с помощью элемента объединения diff_type оно интерпретируется как тип char и ему присваивается значение, равное ASCII символу 'A'.

В строках 7, 11, 14, 17 вызывается функция форматированного вывода printf, которая выводит одну и ту же память объединения diff_type как значения разного типа. Следует обратить внимание, в строке 17 последний

элемент шаблона вывода заменен на %u, что указывает на необходимость вывести значение usi как значение беззнакового типа.

В строках 9, 10, 13, 16 выполняется присвоение diff_type различных значений, интерпретируемых по правилам разных типов.

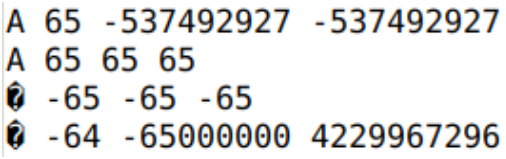


Рисунок 2 — Отображение потока вывода программы

На рисунке 2 представлено отображение потока вывода описанной ранее программы.

Первая строка вывода сформирована строкой 7 исходного текста программы. До этого вывода объединение было интерпретировано как тип char и присвоено значение 65. Первый символ A соответствует выводу diff_type.ch как типа char. Для получения кода символа в шаблоне вывода необходимо указать вывод типа %d. Таким образом получено значение 65, соответствующее символу A английского алфавита. Вывод diff_type.si и diff_type.usi дает произвольные значения, т. к. при инициализации объединения типом char не была задействована вся память объединения. Произвольное значение является результатом вывода мусора из не инициализированной части объединения. При выводе значения как знакового и беззанакового получен одинаковый вывод, т. к. указан один и тот же шаблон вывода %d.

Вторая строка вывода сформирована после того, как все биты объединения сначала были обнулены, а уже затем было присвоено значение 65 как типу char. В этом

случае, при любой интерпретации, значение равно 65, т. е. коду символа A.

Третья строка вывода сформирована после присвоения dif_type значения -65 типа unsigned int. Код -65 был интерпретирован как непечатаемый символ и отображен символ псевдографики. Значения всех остальных типов выведены верно.

Последняя строка вывода сформирована после присвоения памяти объединения значения -65000000. Данное значение не может вместится в один байт типа char. В результате -65000000 было интерпретировано как символ псевдографики и значение -64 типа char. Значение выведено правильно при использовании типа int и неверно при использовании типа unsigned int. Следует обратить внимание, присвоение было приведено именно как тип unsigned int (строка 16 исходного текста программы). Компилятор сделал исполняемый файл, который проверил допустимый объем ОЗУ для хранения значения и просто поместил в них переданные ему байты, без учета типа данных.

При использовании типов данных с плавающей запятой, «эффекты» неверной интерпретации типов данных будут другими. Рассмотрим пример исходного текста программы на рисунке 3.

В строке 4 объявлено объединение с идентификатором diff_type, которое может быть интерпретировано как тип int, float или double.

В строках 7, 10, 13 использована функция форматированного вывода, выводящая память объединения как разные типы данных.

В строках 6 и 9 производится присвоение значения экземпляру объединения deff_type разного типа.

В строке 12 объединение интерпретируется как целое число и ему присваивается значение этого же

объединения как типа double. При этом выполняется неявное приведение присваиваемого значения к целочисленному типу.

02.c

```c
#include <stdio.h>

int main () {
    union { int i; float fl; double dl; } diff_type;

    diff_type.fl = 100.0;
    printf("%d %lf %lf\n", diff_type.i, diff_type.fl, diff_type.dl);

    diff_type.dl = 100.0;
    printf("%d %lf %lf\n", diff_type.i, diff_type.fl, diff_type.dl);

    diff_type.i = diff_type.dl;
    printf("%d %lf %lf\n", diff_type.i, diff_type.fl, diff_type.dl);

    return 0;
}
```

Рисунок 3 — Исходный текст программы с данными с плавающей запятой

Отображение потока вывода программы, полученной из этого исходного текста программы, представлено на рисунке 4.

```
1120403456 100.000000 0.000000
0 0.000000 100.000000
100 0.000000 100.000000
```

Рисунок 4 — Отображение потока вывода программы

Первая строка вывода сформирована 7 строкой исходного текста программы. Объединению при этом присвоено значение 100.0 типа float. При этом

правильное значение сохраняется только в случае, если объединение интерпретируется как тип float. Первое и второе значение отлично от правильного.

03.c

```c
#include <stdio.h>

int main () {
    double dbl1 = 12345.123456;
    double dbl2 = 12345.128456;

printf("%lf\n",dbl1);
printf("%lf\n",dbl2);

    int temp = (int)((double)dbl1*(double)100.0);

printf("%d\n",temp);

    double dbl1_temp = (double) temp/100;

printf("%lf\n",dbl1_temp);

    temp = (int)((double)dbl2*(double)100.0);

printf("%d\n",temp);

    double dbl2_temp = (double) temp/100;

printf("%lf\n",dbl2_temp);

    if (dbl1_temp == dbl2_temp) {
        // Переменные dbl1 и dbl2 равны
        puts("Значения равны");
    }

    return 0;
}

```

Рисунок 5 — Исходный текст программы с явным приведением типов

Вторая строка вывода сформирована 10 строкой исходного текста программы. В этот раз присвоено значение 100.0 как тип double. Правильно отобразить его

получилось только как тип double.

Последняя строка вывода сформирована 13 строкой исходного текста программы. Объединению был присвоен результат неявного приведения значения типа double к типу int.

Неявное преобразование может приводить к неожиданным результатам. При последнем выводе значения типа int и типа double совпали, однако это результат работы в состоянии «неопределенного поведения», описанного в стандарте языка программирования. Не стоит использовать такие способы программирования в надежном приложении. Это может привести, например, к делению на ноль и вызвать аварийное завершение работы программы.

Рассмотрим пример исходного текста программы с явным преобразованием типов, используемого для округления дробных чисел до сотых долей числа. Данный текст программы приведен в предыдущем разделе. Здесь он дополнен отладочными вызовами функции printf в строках 7, 8, 12, 16, 20, 24 для отображения вычисляемых значений после каждой операции. Вызов функции printf специально сдвинут влево для упрощения визуального восприятия. В местах выполнения явного преобразования типов данных, желаемый тип данных указан в скобках. На рисунке 6 показаны данные потока вывода программы, исходный текст которой представлен на рисунке 5.

```
12345.123456
12345.128456
1234512
12345.120000
1234512
12345.120000
Значения равны
```

Рисунок 6 — Отображение потока вывода программы

Первая и вторая строка вывода сформированы 7 и 8 строками исходного текста программы. На них отображаются значения переменных, отличающихся в тысячных долях.

Третья строка вывода содержит округленное значение без дробной части. Этот вывод получен вызовом функции в 12 строке исходного текста программы. Выполнение отбрасывания дробной части показано в 10 строке. Для этого достаточно перед знаком присвоения явно указать требуемый тип (int) .

Четвертая и шестая строки вывода сформированы 16 и 24 строками исходного текста программы. Они выводят округленные значения, сохраненные в переменных dbl1_temp и dbl2_temp соответственно.

Последняя строка вывода сформирована 24 строкой исходного текста программы. Надпись «Значения равны» подтверждает, что округление до сотых переменных dbl1 и dbl2 произведено верно.

Приведенные выше примеры демонстрируют разное расположение данных в ОЗУ и способы их интерпретации в зависимости от типа данных. При использовании указателей разных типов, также необходимо выполнение операций приведения типа данных. Наиболее популярная платформа x86 использует одинаковый размер памяти для большинства указателей разных типов, что приводит к невниманию программистов к контролю размера указателей различных типов. Это может приводит к появлению ошибок в программах при написании исходных текстов программ, переносимых на другие платформы.

Приведем примеры явных приведений для составных типов.

*int *array_ptr = (int []){4, 5};*

Данное выражение объявляет указатель с идентификатором array_ptr на тип int, который будет указывать на массив из двух элементов со значениями 4 и 5.

*const float *array_ptr = (const float []){1.0, 1.1, 1e2};*

В приведенном выражении объявляется указатель на тип float с идентификатором array_ptr, который будет указывать на массив из трех элементов. Следует обратить внимание, последнее значение инициализируется через значение равное $1*10^2$.

Вопрос совместимости типов рассмотрен в пункте 6.2.7 стандарта языка программирования Си. Также в этом пункте приведены ссылки на другие разделы языка программирования, в которых описаны дополнительные правила совместимости спецификаторов, квалификаторов и объявлений. В соответствии с этими правилами один тип может сводится к другому, однако это относится к указателям на массивы и функции.

Для более детального изучения расположения элементов типов данных в ОЗУ можно использовать объединения, включающие в себя массив типа char, с помощью которого будут выводиться изменения в ОЗУ.

Неверное использование приведения типов дает сложно исправляемые ошибки, некоторые из которых удается обнаружить только во время длительной эксплуатации.

5.3. Логические, битовые операции, операции сдвига, операции сравнения

Логические, битовые операции и операции сдвига применяются только к целочисленным значениям!

Базовых логических операций три: отрицание (!), логическое И (&&), логическое ИЛИ (||).

Логическое отрицание упоминается в стандарте языка программирования Си один раз в разделе 6.5.3.3. Логическое И описано в разделе 6.5.13, а логическое ИЛИ в разделе 6.5.14.

```
char value1 = 2;
printf("%d %d %d\n", value1, !value1, !(!value1));
```

Приведенные выше строки текста на языке Си сформируют следующий вывод

```
2 0 1
```

Первая цифра 2 соответствует значению переменной value1 до выполнения операций. Цифра 0 соответствует логической инверсии (отрицанию) числа 2, т. к. любое значение, отличное от 0, в Си является логической единицей. Последняя цифра равна 1, т. к. инверсия от 2 равна 0, а инверсия 0 равна 1.

Следующие строки демонстрируют применения логических И и ИЛИ.

```
char value1 = 2;
char value2 = 0;
printf("%d && %d = %d\n", value1, value2, value1 && value2);
printf("%d || %d = %d\n", value1, value2, value1 ||
```

value2);

В результате их исполнения получен следующий вывод

2 && 0 = 0
2 || 0 = 1

Первая строка вывода демонстрирует результат логической операции И между значениями 2 и 0. Результатом операции является 0. Логическая операция И возвращает 0 или 1. Единица возвращается только в том случае, если оба операнда отличны от 0. В приведенном примере второй операнд равен 0, значит и результат операции равен 0.

Вторая строка вывода является результатом логической операции ИЛИ. Результат логической операции ИЛИ также 0 или 1. Ноль операция возвращает в случае, если два операнда равны 0, во всех остальных случаях, возвращается 1. Так как один из операндов не равен 0, то результат операции 1.

Следующими будут рассмотрены операции сдвига. Они описаны в разделе 6.5.7 стандарта языка программирования Си. К ним относятся две операции:

- операция сдвига влево << ;
- операция сдвига вправо << .

Операция сдвига влево заключается в перемещении всех значений бит на один байт влево и дописывание 0 в крайний правый бит. Значение крайнего слева байта теряется. Арифметически данная операция соответствует умножению на 2.

Операция сдвига вправо выполняется наоборот. Значения всех битов сдвигаются вправо с потерей крайнего правого бита. Арифметически данная операция

соответствует делению на 2.

Ниже приведен пример использования операций сдвига

```
value1 = 3;
printf("%d << 2 = %d\n", value1, value1 << 2);
value1 = value1 << 2;
printf("%d >> 2 = %d\n", value1, value1 >> 2);
value1 = value1 >> 2;
```

Приведенные операции приводят к следующему выводу.

```
3 << 2 = 12
12 >> 2 = 2
```

Выполняя сдвиг влево на два бита значения 3 (00000011), получено значение 12 (00001100). Обратный сдвиг вправо на два бита приводит к преобразованию 12 (00001100) обратно к значению 3 (00000011).

Далее рассмотрим битовые операции. Битовые (побитовые) операции допустимо применять как к битовым полям, так и к значениям других типов. Битовые операции и операции сдвига могут работать неочевидно для программиста при переносе на архитектуры с другой разрядностью или просто другие архитектуры микропроцессора. Часто это связано с неуемной привычкой некоторых программистов строить нагромождения битовых выражений там, где можно обойтись вполне читаемыми выражениями без потери производительности. В связи с этим рекомендуется выносить все подобные операции в отдельные платформозависимые модули.

Побитовое отрицание (~) упоминается в стандарте языка программирования один раз в разделе 6.5.3.3. Операция битовое И (&) рассматривается в разделе 6.5.10. Операция битовое ИЛИ (|) описана в разделе 6.5.12. Также присутствует операция исключающего ИЛИ (^), описанная в разделе 6.5.11.

Рассмотрим операцию отрицания с помощью приведенного ниже исходного текста на языке программирования Си.

```
unsigned char value1 = 1;
unsigned char value2 = 0;
printf("%d %d %u\n", value1, ~value1, ~value1);
printf("%d %d %u\n", value2, ~value2, ~value2);
```

Следует обратить внимание, что результат отрицания выводится сначала как знаковое целочисленное значение, затем как беззнаковое значение.

В результате выполнения приведенного выше исходного текста, будет сформирован следующий вывод.

```
1 -2 4294967294
0 -1 4294967295
```

Первая строка вывода соответствует инверсии значения 1. В случае вывода инверсии 1 в виде знакового целого числа получено значение, соответствующее -2, которое, в свою очередь, соответствует значению на единицу меньше максимально допустимого значения.

Вторая строка вывода соответствует инверсии числа 0. Инверсии 0 соответствует знаковое целочисленное значение -1 или максимально допустимое значение для целочисленного беззнакового типа.

Для демонстрации работы битовых операций

рассмотрим результат работы следующего исходного текста на языке Си

```
printf("1 & 3 = %d (%u)\n", 1 & 3, 1u & 3u);
printf("1 | 3 = %d (%u)\n", 1 | 3, 1u | 3u);
printf("1 ^ 3 = %d (%u)\n", 1 ^ 3, 1u ^ 3u);
```

Ниже показано содержание стандартного потока вывода после выполнения приведенного исходного текста

```
1 & 3 = 1 (1)
1 | 3 = 3 (3)
1 ^ 3 = 2 (2)
```

В результате выполнения первой строки выполнена битовая операция И над числами 1 и 3 как знакового, так и беззнакового типов. В результате получено значение 1, т. к. в значении 1 (00000001) и в значении 3 (00000011) младшие биты равны 1.

Вторая строка выведена как результат битовой операции ИЛИ для тех же значений. В результирующем значении в 1 установлены те биты, которые установлены в 1 в одном из входных операндов.

Третья строка выводит результат битовой операции исключающее ИЛИ. Для отображения всех возможных результатов данной операции ниже приведена таблица истинности этой операции. Вывод таблицы истинности сгенерирован следующими строками текста программы

```
printf("0 ^ 0 = %d\n", 0 ^ 0);
printf("1 ^ 0 = %d\n", 1 ^ 0);
printf("0 ^ 1 = %d\n", 0 ^ 1);
printf("1 ^ 1 = %d\n", 1 ^ 1);
```

В результате получены следующие значения:

0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1
1 ^ 1 = 0

В соответствии с таблицей, значение 1 будет получено только в том случае, если один их операндов равен 0, а другой 1.

Операторы присвоения описаны в разделе 6.5.16 стандарта языка программирования Си. К ним относятся как собственно оператор присвоения, так и сокращенные формы записи выражений с использованием присвоения. Перечислим эти операторы

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

Оператор присвоения используется почти в каждом исходном тексте программы и сложностей с его применением не возникает. Сокращенные формы встречаются реже, но также пользуются популярностью у программистов. Их преимущество в том, что они содержат как минимум на одну ассемблерную инструкцию меньше, чем полная форма.

Например:

x += 3;
x -= 2;
x ^= 1;

соответствует

x = x+3;

```
x = x - 2;
x = x^1;
```

Совместно с операциями присвоения стоит упомянуть операции эквивалентности, которые описаны в разделе 6.5.9 стандарта языка программирования Си и состоят из двух операций: проверка на равенство (==) и не равно (!=). Результатом этих операций является значение 1 или 0 типа int.

Операциям сравнения посвящен раздел 6.5.8 стандарта языка программирования. К операциям сравнения относятся операторы меньше (<), больше (>), меньше или равно (<=) и больше или равно (>=).

Наиболее часто битовые операции используются для проверки состояния одного из битов или для изменения порядка следования байт. Для проверки состояния бита используются следующие действия:

- проверка состояния бита;

```
char bitstatus (char data, char position) {
    return ((data & (1 << position)) != 0);
}
```

- инверсия значения бита

```
char bitreverse (char data, char position) {
    return (data ^ (1 << position));
}
```

Для изменения порядка следования байт можно использовать следующие приемы: интерпретировать целое число как объединение с массивом, содержащим элементы типа char, и выполнять сдвиг после присвоения каждого значения или произвести присвоение элементов

массива самому себе, но в обратном порядке.

Для демонстрации преобразования потребуется объявление составного типа данных объединения byte_order.

```
union byte_order {
    unsigned int data;
    unsigned char bytes[8];
};
```

Сами преобразования выполняют две нижеследующие функции:

- переход от архитектуры **little-endian** к текущей архитектуре

```
unsigned int little_endian_to_memory (unsigned int value) {
    int size_in_bytes = sizeof(int);
    int count = 0;
    union byte_order buffer;
    buffer.data = value;
    unsigned int result = 0;
    for ( ; count < size_in_bytes; count++)
        result |= buffer.bytes[count] << count*8;
    return result;
}
```

- переход от архитектуры **big-endian** к текущий архитектуре

```
unsigned int big_endian_to_memory (unsigned int value) {
    int size_in_bytes = sizeof(int);
    int count1 = 0;
```

```
        int count2 = size_in_bytes-1;
        union byte_order buffer;
        buffer.data = value;
        unsigned int result = 0;
        for ( ; count1 < size_in_bytes; count1++, count2--)
              result |= buffer.bytes[count2] << count1*8;
        return result;
}
```

Наиболее правильное и переносимое решение проблемы смены порядка байт — это использование специальных функций ntohl, htonl, ntohs, htons. Они не являются частью стандарта языка программирования и в соответствии со справочной информацией описаны в arpa/inet.h, однако фактически их определение может находится в другом файле.

5.4. Квалифицирующие типы и спецификаторы функций

Квалифицирующие типы не влияют на размер выделяемой памяти, однако помогают компилятору и программисту уменьшать количество ошибок в исходных текстах программ. Они описаны в разделе 6.7.3 стандарта языка программирования и включают в себя следующие ключевые слова

const
restrict
volatile
_Atomic

Квалифицирующий тип _Atomic также описан в пункте 6.7.2.4, const в 6.6, а restrict в 6.7.3.1 стандарта языка программирования.

Квалифицирующий тип const указывает на то, что объект не модифицируемый. Это относится как к данным, так и указателям на данные или элементам структур. Примеры объявления объектов.

- значение по адресу не изменяется, адрес может изменяться;

*const int * I;*
*int const * I;*

- значение по адресу может изменяться, адрес изменяться не может;

*int * const I;*

- значение по адресу и сам адрес изменяться не

могут.

*const int const * I;*
*int const * const I;*

Также константой является любое число или символ. К именованным константам относят значения, определенные с помощью директивы препроцессора #define .

Восьмеричные константы допустимо объявлять с помощью следующей конструкции '\000', где 000 — любые допустимые цифры.

Для обозначения шестнадцатиричных констант используется последовательность '\x00', где 00 — любые допустимые цифры и буквы.

Квалифицирующий тип restrict применим только к указателям и сообщает, что данные по этим указателям располагаются в разных ячейках памяти, т. е. не пересекаются. Примером пересекающихся данных могут служить несколько указателей на разные элементы как подмассивы одного массива. Особенно следует обращать внимание на наличие этого квалифицирующего типа при использовании функций для работы с памятью.

Пример объявления объекта

*int * restrict a;*

Квалифицирующий тип volatile указывает компилятору не выполнять оптимизацию данной переменной, даже в случае отсутствия обращения к ней. Такая необходимость часто возникает в случае потенциальной возможности изменения значения из другого потока. Также этот квалифицирующий тип используется при работе с портами, в прерываниях и в

регистрах, связанных с внешними устройствами. Переменные, используемые в этих операциях, необходимо объявлять с этим квалифицирующим типом. Пример объявления переменной.

volatile int value;

Указатель на эту переменную может быть объявлен следующим образом.

*volatile int * ptr;*
или
*int volatile *ptr;*

Для объявления volatile-указателя на volatile-переменную используется следующая запись

*volatile int * volatile ptr;*

Программисты с небольшим опытом создания многопоточных приложений иногда путают volatile с атомарными операциями. Операции с объектами квалифицирующего типа volatile выполняются так же, как и все остальные операции, их выполнение может быть прервано в любой момент, поэтому их нельзя использовать для синхронизации потоков.

При использовании volatile переменной указатель на неё также должен содержать квалифицирующий тип.

Квалифицирующий тип _Atomic используется с целью избежать состояний гонки данных. При этом состоянии несколько потоков получают доступ к одной и той же области памяти, и в лучшем случае, значение становится неверным, в худшем, программа может аварийно завершиться.

В соответствии со стандартом языка программирования, объекты атомарных типов являются единственными объектами, при использовании которых невозможно состояние гонки данных, т. е. поток не может прерывать процесс изменения значения объекта, пока его состояние изменяет другой поток. Такой эффект достигается путем соблюдения четырех правил, которые описаны в разделе 6.7.3 стандарта языка программирования.

Следует учесть, атомарность существует только для объектов lvalue, т. е. тех, что указываются до знака присвоения. Пример использования данного квалифицирующего типа

*_Atomic const int *ptr;*

Приведенное выше объявление создает указатель на атомарную константу целочисленного типа int. Также в стандарте языка программирования в разделе 7.17 присутствует библиотека stdatomic.h, с помощью которой возможно использование другого вида объявления атомарных объектов. Например, объявить атомарную операцию целочисленного типа можно с помощью приведенной ниже строки.

atomic_int value;

Базовый тип не равен базовому типу с квалифицирующим типом. Следует иметь в виду, что использование квалифицирующих типов требует выполнения приведения типов, в том числе к их базовым типам.

Спецификаторы функций описаны в разделе 6.7.4 стандарта языка программирования. Описание

объявления функций рассмотрено в разделе 6.9.1. К ним относятся два ключевых слова inline и _Noreturn.

Реализация спецификатора inline часто заключается в том, что текст тела функции подставляется в место её вызова, ускоряя выполнение программы.

Пример функции со спецификатором inline.

inline void print_int (int digit);

Спецификатор _Noreturn предписывает функции не производить возврат в точку вызова. Пример функции со спецификатором _Noreturn.

_Noreturn void function_name ();

Не стоит путать такое поведение с возвратом значения пустого типа void. Отличие будет продемонстрировано с помощью двух прототипов функций. В соответствии с первым прототипом, функция возвращает значение типа void, а выполнение программы продолжается с последующего выражения.

void print_int (int digit);

Второй прототип функции объявлен в заголовочном файле setjmp.h.

_Noreturn void longjmp(jmp_buf env, int val);

Из прототипа функции следует, с её помощью выполняется «дальний переход» в другой блок команд, при этом выполнение выражения следующего за точкой вызова этой функции невозможно.

В соответствии с пунктом 7.23 стандарта языка

программирования в заголовочном файле stdnoreturn.h определено макроопределение noreturn эквивалентное _Noreturn.

Далее приведен пример использования функции, которая ничего не возвращает и не передает управление в вызвавшую её функцию.

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

_Noreturn void print_message(char * message) {
   printf("\n%s\n", message);
   exit(0);
};

int main () {
   print_message("Hello");
   return 0;
}
```

Для сборки программы необходимо указывать версию языка программирования

```
gcc 03.c --std=c17
```

В соответствии с приведенным текстом программы, она будет завершаться вызовом функции exit() в функции print_message(), а не оператором return в функции main() .

Другая причина использования _Noretrun — это реализация длинного межсегментного перехода, при котором производится изменение обоих регистров счетчиков, а не только IP.

5.5 Директивы предпроцессора

Директивы предпроцессора являются еще одним средством, позволяющим изменять исходный текст программы или комбинировать набор файлов, используемых для создания исполняемого файла до начала компиляции. Все действия предпроцессор выполняет до этапа компиляции, что позволяет включать в исходный текст программы конструкции, неиспользуемые в языке программирования Си.

К целям использования директив препроцессора относится выбор фрагментов текста программы в зависимости от:

- модели процессора;
- аппаратной конфигурации ЭВМ;
- конфигурации ПО, включая создаваемое ПО;
- версии и конфигурации используемого компилятора.

Представленные цели достигаются с помощью следующих возможностей:

- замены или удаления символов;
- вставки содержимого произвольного файла;
- условной компиляции;
- вывода сообщений о предупреждениях и ошибках;
- задания режимов работы компилятора.

Не стоит использовать средства предпроцессора в следующих целях:

- использовать директиву #define вместо функций во избежания путаницы со скобками;
- изменять архитектуру, логику программы без объективной необходимости;
- закладывать несколько различных архитектур программы;
- с целью корректировки этапов создания

исполняемого файла, создавая, например, единственно возможную последовательность подстановки файлов директивами include;

- создавать сложночитаемые конструкции, определяемые и переопределяемые в разных файлах одного проекта, например, изменять синтаксис языка программирования в некотором выражении;

- в других случаях использования средств предпроцессора не по прямому назначению.

Некорректное использование директив предпроцессора обычно происходит при обнаружении ошибок проектирования ПО на поздних этапах разработки, многолетней поддержке ПО, отсутствии четкого технического задания, добавление нового аппаратного обеспечения или программного окружения, неучтенного на этапе разработки архитектуры ПО.

Директивы предпроцессора описаны в разделе 6.10 стандарта языка программирования.

Если строка начинается с символа # и не содержит других символов, то эта директива считается пустой директивой и не обрабатывается.

```
#
```

Подключение заголовочных файлов. При использовании символов меньше и больше файл должен находится в каталоге include. При использовании двойных кавычек, файл должен находится в текущем каталоге.

```
#include <stdio.h>
#include "02.h"
```

Для объявления констант и выражений используется

директива #define, которая имеет несколько форм записи.

Первая форма записи директивы #define используется для определения глобальных констант и имеет следуюший вид

#define VALUE 100LU

или

#define VALUE2

Значение константы VALUE равно 100 и имеет тип unsigned long int. Тип константы указывается с помощью следующих символов:

- U или u представляет константу в беззнаковой форме (например, 10U);
- F или f позволяет объявить константу типа float (97.2f);
- L или l объявляет тип long (5423L);
- D или d соответствует типу double (59.3E-19D);
- константы в шестнадцатиричном формате начинаются с комбинации символов 0x (0xFFFF);
- константы в восмеричном формате начинаются с нуля, например 023.

Допускается помбинирование типов констант, например 023LD.

Константа VALUE2 объявлена, но значение ей не задано. С точки зрения синтаксиса предпроцессора она считается объявленной. Её можно использовать в директивах предпроцессора без проверки значения.

Вторая форма записи директивы #define позволяет производить замену одной последовательности символов на другую, разделителями между которыми могут быть

один или несколько символов пробела.

#define DIFF(a,b) (a - b)

Приведенная директива позволяет использовать в исходном тексте конструкцию, синтаксически схожую с вызовом функции

int x = DIFF(15,5);

Символ # , используемый в директиве #define , позволяет заключить последовательность символов в двойные кавычки.

#define text_const(value) # value

Используя следующие строки исходного текста, например в функции main()

int x = 4;
printf(text_const(x) "\n");

В стандартный поток вывода будут выведены символы x и переход на новую строку несмотря на то, что идентификатор x соответствует целочисленной переменной. Он передается в выражение text_const, объявленное с помощью директивы предпроцессора #define. Выражение преобразует идентификатор x в строковую константу "x", которая и будет выведена.

С помощью последовательности ## , используемой в директиве #define или другом макросе, производится частичное замещение идентификатора.

*#define SUMM(x,y,z) (a##x + *

```
a##y + \
a##z)
```

Конструкция в приведенном примере позволяет использовать следующий текст программы

```
int a1 = 4, a2 = 2, a3 = 1;
printf("%d\n", SUMM(1,2,3)*2);
```

Второй аргумент функции printf будет заменен на выражение (a1+a2+a3)*2 за счет частичной замены идентификаторов, выполненной с помощью последовательности ##. Также в примере директивы #define используется символ \ , позволяющий переносить элементы синтаксиса директивы на следующую строку.

Другой прием использования ##, который также использует перенос на следующую строку

```
#define object_property( x ) x##X, x##Y, x##Height, \
                             x##Width
```

Объявление позволяет использовать прототип следующего вида

```
int object_property( object );
```

который будет преобразован препроцессором к следующему виду:

```
int object_property( objectX, objectY, objectHeight,
objectWigth);
```

Отключение объявления констант и выражений производится директивой #undef, причем применение

директивы к неопределенному ранее идентификатору не считается ошибкой.

#undef VALUE1

Для проверки на объявление используется директива #ifdef.

```
#ifdef VALUE1
    #define PRINT() puts("VALUE1 define");
#endif
#ifndef VALUE1
    #define PRINT() puts("VALUE1 undefine");
#endif
```

Использование в исходном тексте программы выражения PRINT(); приведет к выводу сообщения "VALUE1 define", если константа VALUE1 определена или "VALUE1 undefine" в обратном случае.

Директивы #ifdef, #ifndef, #if, #elseif, #else, #endif позволяют реализовывать условную компиляцию программы до обработки текста программы компилятором. Каждая директива #if должна заканчиваться директивой #endif. Между ними может использовать любое количество директив #elseif и только одна #else, причем она должна быть последней. Например:

```
#if VALUE == 1
    #define PRINT() puts("VALUE = 1");
#elif VALUE == 2
    #define PRINT() puts("VALUE = 2");
#else
    #define PRINT() puts("VALUE = undefine");
```

#endif

Используя выражение PRINT(); можно определить, какое значение имеет константа VALUE.

Для перехода интерпретации директив предпроцессора в другой файл используется директива #line.

#line 151 "02.c"

Данный пример предписывает перейти в файл 02.c текущего каталога на строку 151. В практическом программировании директива используется только опосредовано, через другие директивы, например #include.

Для объявления токенов и их состояний используется компиляторозависимая директива #pragma.

#pragma listing on "..\dir"

Перевести некоторое абстрактное значение listing в состояние on для объекта dir, расположенного в текущем каталоге.

Директива #error используется для выдачи диагностических собщений.

Для расширения возможностей при использовании предпроцессора рекомендуется ознакомится с автоматически создаваемыми константами, описанными в разделе 6.10.8 стандарта языка программирования.

В качестве общепризнанного примера грамотного использования средств предпроцессора можно рассматривать заголовочный файл queue.h из каталога /usr/include/sys/, который не является частью стандарта

языка программирования. Объявленные в нем составные типы и функции реализуют связные списки и эта реализация признается эталонной на протяжении нескольких десятилетий.

5.6 Отличия между стандартами языка Си разных лет

История появления стандарта языка программирования Си описана в Википедии достаточно подробно, чтобы получить представления об основных этапах его развития. В различных книгах встречается дополнительная информация, уточняющая общепринятую историю или личные переживания участников процесса зарождения языка Си.

Язык продолжает развиваться и сейчас. Разработчики компиляторов не всегда успевают за нововведениями, публикуемыми в новых стандартах. По этой и другим причинам скорее всего не существует ни одного компилятора, полностью реализующего текущий стандарт Си. При этом в ближайшие годы должен появится следующий стандарт, с еще большим количеством возможностей.

Во времена зарождения языка, значительная часть времени разработки программы тратилась на описание операций с ОЗУ, автоматизацию работы с регистрами микропроцессора. Несмотря на различия в архитектурах микропроцессоров, существует уровень абстракции этих операций, единый для всех основных архитектур микропроцессоров. Именно это и стало фундаментом языка Си. Эта часть языка монолитна и не меняется на протяжении всего времени его существования. Остальные части стандарта могут подвергаться изменениям, поэтому крайне рекомендуется при проектировании программного обеспечения выделять все взаимодействия с любым аппаратным обеспечением, многопоточность, сетевые взаимодействия в отдельные модули или функции, а логику и математику программы выделять в её ядро. Таким образом можно написать

исходный текст, который может оставаться неизменным десятки лет, а изменению будут подвергаться только отдельные модули взаимодействия с другими элементами информационной системы.

Изучая новые стандарты, не стоит забывать о миллионах строк исходных текстов других программ, которые, возможно, придется использовать при написании своей программы. В связи с этим, любое новшество или новый прием написания, используемые в исходном тексте, должны обосновываться, а не браться за основу по причине того, что оно уже появилось. Данное правило особенно важно для языка Си, т. к. многие специализированные микропроцессоры используют урезанные возможности языка. В связи с этим, чем меньше разных дополнений будет содержать исходный текст, тем меньше придется переписывать при переходе на новую платформу.

Наибольшее количество важных изменений было закреплено в стандарте C99 (1999 г.). Стандарты более раннего периода в практической работе можно не принимать во внимание. Следующей важной вехой стал стандарт C11 (2011 г.), наиболее важными нововведениями которого стала библиотека многопоточности, функции определения смещения, улучшенная поддержка UNICODE. Стандарт C17 (2017 г.) имеет также ряд дополнений, но они относятся к деталям и мало влияют на написание исходного текста. Часто его представляют как черновик стандарта 2018 года. Стандарт C18 (2018 г.) закрепил за собой имидж уточняющего документа. Следует иметь в виду, текущий стандарт языка программирования продается по достаточно высокой цене, но в свободном бесплатном доступе имеется последний черновик предыдущего стандарта. Информация содержащаяся в черновике,

достаточна для написания программ, а финальный вариант стандарта представляет большой интерес скорее для разработчиков компилятора.

5.7. Переменные окружения. Потоки ввода-вывода. Передача параметров

В соответствии со стандартом прикладного программного интерфейса для ОС POSIX, при создании процесса, ОС предоставляет служебную информацию для каждого создаваемого процесса в виде специальных значений, называемых переменными окружениями. Для обеспечения обмена информацией с создаваемым процессом ОС регистрирует три участка памяти, которые используются для обмена информацией с этим процессом. Доступ к этим участкам памяти осуществляется с помощью абстракций потоков ввода, вывода и потока ошибок. С целью передачи параметров, используемых при запуске программы, реализуется механизм указания ключей после названия процесса.

Поток ввода имеет обозначение stdin и имеет номер 0. Поток вывода имеет обозначение stdout и имеет номер 1, а поток ошибок имеет обозначение stderr и имеет номер 2.

Рассмотрим учебный пример использования всех трех механизмов с помощью программы, исходный текст которой представлен на рисунке 1. Программа получена из этого исходного текста с помощью команды

gcc 01.c && ./a.out in.txt out.txt error.txt

Переданными параметрами являются последовательности символов: in.txt, out.txt и error.txt. Далее следует описание строк исходного текста программы, поясняющих механизмы, описываемые в данном разделе.

В строке 4 используется прототип главной функции main, принимающей три параметра. Параметр argc

содержит числовое значение, равное количеству переданных параметров, строкой вызова программы. Следует учесть, по стандарту POSIX первый параметр, передаваемый в программу всегда равен имени исполняемого файла программы. В связи с этим, минимальное значение argc равно 1, даже в том случае, если в строке вызова отсутствуют передаваемые значения.

Аргумент argv является двумерным массивом, содержащим строки, в каждой из которых находится по одному переданному параметру. Соответственно, в строке argv[0] всегда находится название исполняемого файла программы.

Аргумент env содержит двухмерный массив. В нем каждая строка соответствует некоторой переменной окружения, которые были созданы ОС при создании процесса. Последней строкой этого массива является строка, не содержащая символов.

В строке 5 выполняется проверка количества переданных параметров. Их должно быть ровно 4. Такая проверка необходима, т.к. в строке 7 выводится список переданных аргументов и если их количество не совпадет, ОС завершит программу из-за нарушения её адресации.

В строке 6 производится вызов функции system() из стандартной библиотеки stdlib. Аргумент функции — это строка с командой "pwd > 1.txt". Выполнение данной команды в ЭТ приведет к выводу текущего каталога в стандартный поток вывода, а символ ">" перенаправляет стандартный поток вывода программы pwd в файл 1.txt. Вызов функции system приводит к тому, что команда, переданная в неё, исполняется так же, как и при её выполнении в ЭТ.

01.c

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

int main (int argc, char *argv[], char *env[]) {
    if (argc != 4) return 0;
    system("pwd > 1.txt");
    fprintf (stdout, "Переданные параметры: %s, %s, %s, %s\n",
            argv[0], argv[1], argv[2], argv[3]);

    system("env > 1.txt");
    stdin = freopen(argv[1], "r", stdin);
    stdout = freopen(argv[2], "a", stdout);
    stderr = freopen(argv[3], "a", stderr);

    char buffer[100] = {0};
    while (fscanf(stdin, "%s", buffer) != EOF) {
        fprintf(stdout, " %s", buffer);
        fprintf(stderr, " %s", buffer);
    }

    int count = 0;
    while (env[count] != NULL) {
        printf("%s",env[count]);
        puts("");
        count++;
    }

    system ("DISPLAY=:0 nohup notify-send \"Сообщение\" \"От программы\"");
    system ("ssh user@localhost 'DISPLAY=:0 nohup notify-send \"Сообщение\" \"От программы\"'");

    return 0;
}
```

Рисунок 1 — Использование переданных параметров, перенаправление стандартных потоков и переменных окружения

В строках 7-8 описан вызов функция fprintf. Её первым аргументом является поток, в который будет выведено форматированное сообщение. Второй аргумент соответствует шаблону вывода операторов printf, scanf. Начиная с третьего аргумента идет список переменных, значения которых будут подставлены и отформатированы

в соответствии со спецификаторами типов шаблона вывода. В данном случае, в стандартный поток вывода выводятся имена переданных в программу параметров.

В строке 10 вызывается функция system(), которая выполняет команду "env > 1.txt". В команде для перенаправления потока вывода используется символ ">", что приведет к перезаписи файла 1.txt строками, содержащими переменные окружения программы.

В строке 11 вызывается функция переоткрытия стандартного потока ввода stdin в режиме на чтение в поток, соответствующий файлу, имя которого передано аргументом с индексом 1. После выполнения этого вызова, в поток ввода программы будет помещено все содержимое файла in.txt.

В строке 12 вызывается функция переоткрытия, которая перенаправит стандартный поток вывода в файл, имя которого содержится в аргументе с индексом 2 в режиме добавления. Все данные, отправляемые в стандартный поток вывода будут записываться в файл out.txt.

Выполнение вызова строки 13 приведет к записи всех данных стандартного потока ошибок в файл error.txt.

В строках 16-19 производится считывание стандартного потока ввода, присоединенного к файлу in.txt до конца файла и запись считанных значений в стандартный поток вывода и стандартный поток ошибок.

В строках 22-26 производится вывод всех значений переменных окружения в стандартный поток вывода, при этом соединение потока вывода с файлом out.txt сохраняется.

В строке 28 выполняется вызов команды, приводящей к отображению информационного сообщения в графическом интерфейсе. Строка 28 перенесена на следующую строку. Продолжение строки

можно отличить по отсутствию номера строки в её начале. Следует обратить внимание, в этой строке используются символы двойных кавычек в шаблоне вывода. Для их корректной интерпретации необходимо использовать правила экранирования символов.

В строке 29 производится вызов той же команды, что и в строке 28, но с использованием транспортного сетевого протокола удаленного управления ssh. Для её успешного выполнения необходимо настроить безключевой доступ на устройство, сетевой адрес которой будет указан после символа @. В этом случае сообщение в графическом интерфейсе отобразится на другом устройстве.

До выполнения программы на накопителе присутствовали файлы in.txt следующего содержания

first string
second string

и файл программы.

После создания и исполнения программы с помощью указанной выше команды будут созданы файлы out.txt и error.txt. Они будут созданы ЭТ.

В ЭТ будет выведена строка

Переданные параметры: ./a.out, in.txt, out.txt, error.txt

Строка выводится функцией fprintf, описанной в строках 7-8. На момент исполнения строк 7 и 8 стандартный поток вывода еще проассоциирован с терминалом и выводит сообщения в него.

В графический интерфейс будет выведено графическое сообщение, представленное на рисунке 2.

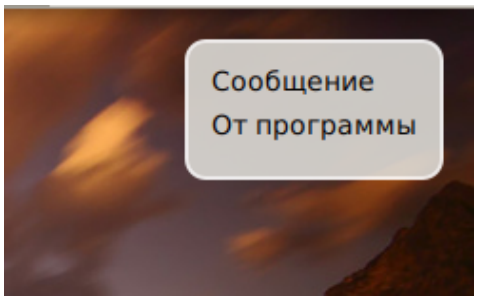


Рисунок 2 — Графическое сообщение

Содержимое файла in.txt не изменилось, т.к. доступ к нему предоставляется только для чтения.

Файл 1.txt, созданный строкой 6 текста программы и пересозданный строкой 10, содержит все переменные окружения. Приведены три первые переменные окружения.

LESSKEY=/etc/.less
XDG_VTNR=1
XDG_SESSION_ID=2

Файл out.txt открыт в строке 12, а запись в него производится в строках 17 и строках 23, 24. В строке 17 явно указано о записи в стандартный поток вывода содержимого буфера. В строках 23, 24 используются операторы printf и puts, которые выполняют вывод в стандартный поток вывода. Все записи в стандартный поток вывода, выполняемые после 12 строки, приводят к записи в файл с именем, переданным argv[2], а именно out.txt. Последовательность "first string second string" перенесена из файла in.txt с помощью строки 17. Остальные данные перенесены строками 23, 24.

Первые строки файла out.txt

first string second stringXDG_VTNR=1

LESSKEY=/etc/.less
SSH_AGENT_PID=743
XDG_SESSION_ID=2

Содержимое файла error.txt

ssh: connect to host localhost port 22: Connection refused
first string second string

Наиболее интересная ситуация сложилась с содержимым файла error.txt. Первая строка сообщает о невозможности подключится по протоколы ssh и выполнить команду. Данное сообщение об ошибке сформировано программой ssh в ответ на её вызов в строке 29. Однако вторая строка сформирована строкой 18, которая переносит содержимое стандартного потока ввода в стандартный поток ошибок, т.е. раньше. Это стало возможно в связи с тем, что данные потока ошибок накапливались в буфере программы и были записаны в файл error.txt только после завершения программы. Данные же потока ошибок ssh были записаны в файл раньше, т.к. работа с ssh завершена в 29 строке. Для принудительного сброса буфера в файл используется специальная функция FLASH.

Работая с ЭТ, также можно использовать подобные приемы. ЭТ также является процессом, а значит имеет переменные окружения и для него создаются стандартные потоки.

Для отображения переменных окружения используются команды printenv или env, возможности которых отличаются доступными ключами. Первые три строки вывода этой команды соответствуют выводу значений строк, полученных с помощью аргумента

функции main.

XDG_VTNR=1
LESSKEY=/etc/.less
SSH_AGENT_PID=743

Примером перенаправления стандартных потоков могут служить следующие команды. Для их демонстрации необходимо запустить два ЭТ. В одном ЭТ необходимо выполнить команду

tty

Выводом команды является номер текущего ЭТ.
Примерный вывод

/dev/pts/1

Во втором ЭТ выполняется команда (после "1" пробел отсутствует)

echo “Строка улетела в другой ЭТ” 1>/dev/pts/1

В результате выполнения команды в первом ЭТ будет выведено сообщение

Строка улетела в другой ЭТ

6. Технологии программирования. Средства автоматизации разработки

6.1. Жизненный цикл ПО. Методологии разработки ПО

Жизненный цикл программного обеспечения начинается в момент осознания потребности в нем и заканчивается в момент вывода всех его элементов из эксплуатации.

Существует ряд стандартов, используемых для разработки ПО. К ним относятся как стандарты, описывающие жизненный цикл программного обеспечения, так и стандарты, посвященные разработке средства автоматизации или специализированные стандарты для разработки ПО с целью применения в областях жизнедеятельности, связанных с высоким риском или особой ценностью.

К стандартам, описывающим жизненный цикл программного обеспечения, относятся:

- ГОСТ Р ИСО/МЭК 12207-2010 Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств;
- ISO/IEC 12207:2008 «System and software engineering — Software life cycle processes»;
- ГОСТ 34.601-90 Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания;
- и другие.

К соответствию внутренних регламентов разработки общепринятым стандартам разработки стремятся только крупные компании, в которых количество разработчиков может исчисляться сотнями и тысячами.

Сегодня большинство популярных программ разрабатывается небольшими компаниями и малыми

группами разработчиков, в связи с этим, стандарты разработки ПО могут показаться устаревшими и ненужными. Однако существует множество областей разработки, где малыми силами решить поставленные задачи невозможно.

При небольших трудозатратах и в небольших проектах нет необходимости проходит все этапы жизненного цикла программного обеспечения, что приводит к значительной экономии средств и ускорению разработки. Такой взгляд на разработку прижился даже в крупных корпорациях и привел к новой идеологии разработки, которую упрощенно можно назвать постоянная бетта версия. В конкурентной борьбе упор делается на скорость выпуска продукта, часто в ущерб его качеству. Кроме потери в качестве продукта игнорирование стандартов разработки приводит к проблемам с поддержкой и дальнейшим развитием проекта, что в свою очередь приводит к удорожанию поддержания продукта в актуальном состоянии длительное время.

Вместо соблюдения стандартов жизненного цикла ПО сегодня используют различные методологии разработки, которые не охватывают весь жизненный цикл ПО. Почти все они пришли из англоговорящих стран. На сегодняшний день их существует несколько десятков. Перечислим некоторые из них:

- Waterfall — традиционный подход, каскадная модель, водопад;
- Incremental Model — инкрементная модель;
- Iterative Model — итерационная модель;
- Spiral Model — спиралевидная модель;
- Rational Unified Process (RUP)— рациональная модель;
- Agile — общая методология гибкой разработки;

- Crystal Clear — методология с уравниванием разработчиков в коллективе;
- Dynamic Systems Development Model (DSDM) – динамическая модель;
- Feature Driven Development (FDD) — методология, рассматривающая будущие изменения;
- Joint Application Development (JAD) — ориентированный на пользователя подход;
- Rapid Application Development (RAD) — модель быстрой разработки;
- Scrum — концепция работы в условиях сорванных сроков и идеологического кризиса;
- Test-Driven Development (TDD) — разработка через тестирование;
- Lean Development (LD) — метод, предполагающий бережное отношение ко всем участникам процесса;
- Extreme Programming (XP) — экстремальная разработка в динамической среде.

Информация по всем методологиям разработки доступна в свободном доступе. У каждой методологии есть свои преимущества и ограничения. Каждой из них можно посвятить главу книги. На практике выбор методологии зависит от многих факторов, включая предпочтения конкретного руководителя. Критическим случаем является экстремальное программирование, смысл которого сводится к достижению цели любыми способами за минимальное время.

Одним из главных документов жизненного цикла программного обеспечения является техническое задание на его разработку. Первой ошибкой, которую совершают все начинающие самостоятельные программисты — отсутствие ТЗ или его низкое качество. В некоторых случаях бывает так, что разработка ведется фактически без него. Также ТЗ может присутствовать в

виде расплывчатого рамочного описания. Причинами этого могут стать:

- низкая квалификация составителя ТЗ;
- выделение недостаточных ресурсов на создание ТЗ;
- вмешательство бюрократической машины;
- необходимость удовлетворения сторонних требований, которые в данном случае ухудшают качество ПО;
- создание ТЗ на основе «рамочной договоренности».

Любое отклонение от равновесия в вопросе стоимости работ, предлагаемая оплата должно вызвать подозрение и привести к дополнительному подробному изучению ТЗ.

Несмотря на все предостережения, разработка без качественного ТЗ возможна. Существует ряд приемов, которые можно использовать в таком случае, повышающие вероятность успешного выполнения проекта. К ним относятся:

- наличие этапности разработки с четко установленными сроками и объемами работ;
- этапы разработки должны заканчиваться оплатой до перехода к следующему этапу разработки;
- объем оплаты этапа должен соответствовать трудоемкости этапа;
- наличие этапа разработки прототипа или полнофункционального прототипа, после утверждения которого дальнейшее изменение ТЗ должно быть запрещено;
- выполнение разработки документации после выполнения демонстрации готового ПО и его утверждения со стороны заказчика;
- четкое определение объема и сроков работ, включенных в гарантийные обязательства;
- четкое разграничение авторских и имущественных

прав на разработанное ПО.

Все, что не вкладывается в указанные правила, должно оплачиваться заказчиком отдельно.

Проблемами, связанными с разработкой ТЗ, архитектуры ПО должны заниматься специалисты со значительным стажем программирования. Для начинающего программиста чаще важнее найти тот самый проект, который поможет ему в начале профессионального пути получить опыт реальной разработки, повысить уровень знаний, ликвидировав имеющиеся пробелы, получить хорошие характеристики и рекомендации. Всего этого нет на «токсических» вакансиях. Одной из причин появления таких вакансий — нечестное поведение предыдущего сотрудника, который, не справляясь со своими обязанностями, тянет до последнего и внезапно бросает проект непосредственно перед его сдачей заказчику. Приведем некоторые признаки «токсической должности»:

- длинный список требований к должности;
- должностные обязанности, не соответствующие специальности (погрузка, монтаж оборудования, подготовка налоговой отчетности);
- использование избыточного числа технологий (список, содержащий более 5 технологий);
- требования не соответствуют вопросам и заданиям на собеседовании (ищут специалиста высшей категории, дают решать школьные задачки или задачи из других областей);
- «случайный» состав людей, проводящих собеседование;
- слабая заинтересованность в приеме нового сотрудника (долго не отвечают, проявляют общую незаинтересованность);
- отсутствие возможности общаться с более

опытными разработчиками;

- отсутствие наставника;
- проблемный непосредственный начальник;
- «серая» заработная плата (признак низкой прибыльности компании);
- сложные схемы начисления заработной платы, бонусы в разы превышают размер заработной платы;
- негативные отзывы в общедоступных источниках, датированные разными годами.

Такие должности также можно использовать с пользой для себя, если планируется работать в компании продолжительное время. Они позволят зарекомендовать себя как настоящего бойца, не боящегося трудностей. При этом велик риск заработать негативную репутацию как специалиста.

Перейдем непосредственно к этапам написания исходного текста программы. Эти действия выполняет непосредственно программист.

1. Формулирование задачи на основе ТЗ или распоряжения непосредственного руководителя.
2. Подготовка к написанию исходного текста программы (разбор технологий, библиотек, архитектур, которые планируется использовать, разработка алгоритмов, фактическое нахождение файлов с требуемыми готовыми фрагментами исходных текстов и т.д.).
3. Определение времени, необходимого на разработку программы или функции.
4. Создание файловой структуры, каталогов, файлов, необходимых для разработки.
5. По необходимости инициировать средство контроля версий, например git, который будет описан в разделе 6.2 этой главы.
6. По необходимости определится с использованием

средств автоматизации.

7. Определить цели использования средств предпроцессора.

8. Перейти к написанию текста программы:

- описать все функции и требуемые действия в них обычным текстом на родном языке, желательно включать в описание ключевые слова средства автоматизации документирования (раздел 6.5. этой книги);

- выполняется написание текста программы, при этом сделанное ранее описание становится комментариями к тексту программы.

Таким образом, сначала будут написаны комментарии к исходному тексту программы, а уже затем сам текст программы.

6.2. Система контроля версий git

На протяжении нескольких десятилетий обсуждается возможность безфайлового хранения информации. Развитием этой идеи является технология «машина времени» корпорации AppleTM и постепенный переход к безфайловому хранению в мобильной ОС Андроид. Несмотря на все попытки, иерархическое хранение информации сейчас является единственным универсальным общеиспользуемым способом хранения информации. Принцип иерархического хранения реализуется в первую очередь файловой системой информационных систем.

Наиболее удачной комбинацией двух подходов иерархического способа хранения информации и безфайлового хранения является система контроля версий git. Принцип её работы заключается в том, что она хранит отличия от предыдущего зафиксированного состояния, а не все версии всех файлов.

Хранилище в терминологии git — репозиторий. **Git репозитории** — это каталог, содержащий как пользовательский набор файлов и каталогов, так и служебный каталог системы контроля версий с именем .git, обычно имеющим свойство «скрытый».

Система git имеет сетевую архитектуру, однако в ней не предусмотрен выделенный сервер. Все репозитории на разных ЭВМ равнозначны. При наличии соответствующих прав любой узел может отдать изменения или получить их с любого узла. Кроме того, при локальном применении git доступны все его функции.

К основным операциям, применимым к репозиториям, относятся: инициализация репозитория, отправить изменения, получить изменения,

клонирование репозитория.

Для работы с системой git необходимо знать следующие понятия.

Репозиторий по-умолчанию (origin) — имя по-умолчанию для репозитория по-умолчанию. Может меняться пользователем.

Изменения — отличия в файловой структуре и содержимом его элементов от состояния во время предыдущей фиксации изменений. Понятие используется для определения наличия неучтенных изменений. Выделяются красным цветом при использовании команды получения состояния репозитория.

Учтенные изменения (index) — изменения, учет которых выполнен с помощью специальной команды. Наличие учтенных изменений позволяет проводить дальнейшие изменения или зафиксировать их в виде «коммита». Выделяются зеленым цветом при использовании команды получения статуса репозитория.

Коммит (commit) — зафиксированные изменения. Является единицей, которыми обмениваются репозитории. Используется в операциях добавить изменения, получить изменения. К ним применимы операции объединения **мердж (merge)** и автоматическое объединение **автомердж (automerge)**. Операции объединения используются при параллельной работе нескольких человек с одним каталогом. Следует обратить особое внимание! Коммит — это изменение, а не состояние, и разрабатывать стратегию обмена коммитами следует именно из этих соображений.

HEAD — указатель на текущий коммит, к которому применимы операции с использованием указателя на «вершину» коммитов. Единственный для текущего состояния.

Ветка — способ логического структурирования

коммитов. Коммиты добавляются к одной из веток, для объединения с коммитами других веток используются операции объединения и автообъединения.

checkout — операция смены текущей ветки и некоторые другие действия.

Ветка по-умолчанию (master) — ветка по-умолчанию, всегда присутствующая в репозитории.

Заначка (stash) — отложенные изменения, которые не входят в состав коммита. Список заначек можно просматривать. Заначки можно применять, отбрасывать.

Рассмотрим некоторые приемы работы с системой git.

В первую очередь необходимо создать каталог проекта, например example

mkdir example

Следующим шагом необходимо сделать созданный каталог проекта текущим

cd example

Далее выполняется инициализация локального репозитория git командой

git init

После её успешного выполнения будет выведено сообщение

Initialized empty Git repository in example/.git/

После выполнения инициализации репозитория необходимо задать информацию о пользователе. Эти

команды приведены в конце этого раздела.

Также возможна инициализация проекта с помощью клонирования уже существующего репозитория, выполнив команду

git clone пользователь@имя ЭВМ с репозиторием:путь до каталога репозитория

При выполнении локального клонирования репозитория необходимо учитывать, по умолчанию создается жесткая ссылка, а не клонирование репозитория. Создание полноценного клонированного репозитория выполняется с помощью ключа --no-hardlinks.

Замечание! Если репозиторий содержит несколько веток, сначала клонируется главная master. Затем возможен просмотр всех доступных веток с помощью команды

git branch -a

Команда отображает все ветки, включая скрытые.

Сделать активной другую ветку можно с помощью перехода на неё. Переход выполняется командой

git checkout <ИМЯ ВЕТКИ>

или выполнить

git pull --all

После этого можно запросить статус проекта

git status

В случае создания нового репозитория статус проекта будет содержать следующие сообщения

On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)

В каталоге проекта example появился новый каталог .git. Далее в каталоге проекта был создан текстовый файл 01.c, содержащий исходный текст программы на языке Си, приведенный ниже.

```
#include <stdio.h>

int main () {
   return 0;
}
```

Далее выполняется проверка состояния проекта. Перед выполнением команды необходимо убедится о том, что текущим является каталог проекта.

```
git status
```

В результате будет получена следующая информация

On branch master

Initial commit

Untracked files:
(use "git add <file>..." to include in what will be committed)

01.c

nothing added to commit but untracked files present (use "git add" to track)

В соответствии с полученным сообщением система git обнаружила в каталоге проекта новый не отслеживаемый файл 01.c. Этот файл и является изменением в проекте. Для того чтобы система git учитывала изменения в этом файле, необходимо выполнить команду регистрации всех новых объектов

git add -A

После выполнения команды, файл 01.c и его содержимое будут отслеживаться. С этого момента он относится к учтенным изменениям. Далее повторно выполняется следующая команда

git status

Здесь и далее будет приводиться только часть вывода системы git из-за их объема. Сообщение системы git будет содержать следующую строку

new file: 01.c

Строка сообщает что файл 01.c и изменения в нем отслеживаются. Далее нужно создать коммит следующей командой

git commit -m "First commit"

Будет создан коммит с комментарием "First commit". Для задания комментария используется ключ -m. Будет выведено сообщение

1 file changed, 5 insertions(+)
create mode 100644 01.c

Сообщение содержит информацию об изменениях в проекте. Выполняется проверка статуса проекта.

git status

Сообщение подтверждает отсутствие изменений, требующих фиксации в коммите.

On branch master
nothing to commit, working tree clean

Теперь в проекте содержится два коммита. Первый коммит всегда создается при инициализации репозитория, второй был создан с помощью соответствующей команды. Посмотреть существующие коммиты в виде дерева к ЭТ можно с помощью следующей команды.

git log --graph

В результате будет запущена программа отображения сообщения системы git, содержащая два коммита. Для выхода из программы отображения необходимо нажать на клавиатуре клавишу q, при

выбранной английской раскладке.

** commit 61b94c3e8530ecbbb0c9b1994e0c2060952c68eb*
Author: Valerii G. Abashin <valeriy@abashin.ru>
Date: Sun Jan 13 16:51:29 2019 +0300

First commit

Для получения подробной информации об изменениях с помощью программы с графическим интерфейсом необходимо использовать следующую команду.

gitk

или

gitk --all

После первого запуска графический интерфейс будет непонятен, однако после выполнения всех команд, приведенных в этом разделе, неопределенность будет устранена. Второй вариант команды отображает информацию не только текущей, но и всех остальных веток. Команды git status, git add -A, git commit -m "Comment" обычно выполняются друг за другом и являются необходимым минимумом для повседневной работы.

Следующий шаг — внесение изменений в файл 01.c. После третьей строки добавлена указанная ниже строка и файл сохранен

puts("Hello");

Выполнение команды git status приведет к выводу сообщения, содержащего следующую строку.

modified: 01.c

Изменения в файле 01.c определены, но не учтены. Далее выполняются приведенные ранее команды

git add -A
git commit -m «Second commit»
git log --graph

Будет получен следующий вывод системы git

** commit e0b23bf6d8579985dde1970ab919d0839ab6d3b9*
| Author: Valerii G.Abashin <valeriy@abashin.ru>
| Date: Sun Jan 13 17:21:56 2019 +0300
|
| Second commit
|
** commit 61b94c3e8530ecbbb0c9b1994e0c2060952c68eb*
Author: Valerii G. Abashin <valeriy@abashin.ru>
Date: Sun Jan 13 16:51:29 2019 +0300

First commit

Новые коммиты необходимо создавать после выполнения каждой небольшой задачи.

Применение системы git позволяет осуществлять **поиск некоторого слова по шаблону**. Ниже приведен пример поиска слова «puts». Пробел перед словом

показывает что это название функции, а не часть произвольного слова.

git log -S ' puts'

После выполнения команды будет запущен редактор текста в ЭТ со следующим содержанием

commit e0b23bf6d8579985dde1970ab919d0839ab6d3b9
Author: Valerii G. Abashin <valeriy@abashin.ru>
Date: Sun Jan 13 17:21:56 2019 +0300

Second commit

Выход из редактора осуществляется клавишей q с включенной английской раскладкой клавиатуры. Обычно требуется осуществить поиск только названий коммитов. Для этого необходимо дополнить команду следующим образом.

git log -S ' puts' --oneline

В результате будет получен компактный вывод

e0b23bf Second commit

Кроме обычного поиска система git предоставляет множество других возможностей. Например, можно определить, когда была изменена каждая строка исходного текста программы или конфигурационного файла. Для этого можно использовать следующую команду.

git blame -L 3,5 01.c

В результате будет получен вывод строк файла 01.c, до которых будет указан номер коммита, его автор и время её написания.

^61b94c3 (Valerii G. Abashin 2019-01-13 16:51:29 +0300 3) int main () {
e0b23bf6 (Valerii G. Abashin 2019-01-13 17:21:56 +0300 4) puts("Hello");
^61b94c3 (Valerii G. Abashin 2019-01-13 16:51:29 +0300 5) return 0;

Символ ^ указывает на коммит, который является HEAD.

Рассмотрим основные **команды управления заначками**, т. е. информацией зарегистрированной git, но отложенной и неиспользованной для создания коммита.

Для создания заначки необходимо модифицировать файл 01.c. Его модификация заключается в добавлении пятой строки со следующим содержанием.

puts("Stash");

Выполненная после этого команда git status сообщает о наличии неучтенных изменений в файле 01.c. Без отказа от этих изменений или их добавления в новый коммит невозможно выполнение операций с коммитами. Выходом из ситуации является использование заначек. Для начала просмотрим список заначек с помощью команды

git stash list

Их быть не должно. Далее спрячем имеющиеся изменения следующей командой.

git stash

или

git stash apply

После выполнения команды система git выведет следующую информацию

Saved working directory and index state WIP on master: e0b23bf Second commit
HEAD is now at e0b23bf Second commit

Теперь выполнение команды git stash list приведет к отображению следующей информации

stash@{0}: WIP on master: e0b23bf Second commit

Была создана заначка stash@{0}. После этого добавленная строка в файл 01.c будет удалена, но она будет спрятана в заначку и её можно будет вернуть с помощью команд управления заначками.

Далее приведены команды добавления и удаления заначек, которые не следует применять к экспериментальному репозиторию!

При этом вызов команды git status приведет к получению сообщения об отсутствии изменений. Они все были спрятаны в заначку. Созданную заначку можно

использовать разными способами. Например, чтобы применить изменения в заначке по её номеру, необходимо выполнить команду

git stash apply stash@{0}

Для удаления заначки по её номеру используется команда

git stash drop stash@{0}

Для удаления всех заначек следует выполнить команду

git stash clean

Для дальнейшей демонстрации операций с репозиторием example необходимо применить следующую команду, которая, используя заначку с индексом 0, создаст новую ветку с именем dev01

git stash branch dev01

Будет выведена следующая информация

Switched to a new branch 'dev01'
On branch dev01
...
modified: 01.c
...

С этого момента в репозитории существует две ветки: master и dev01. Для отображения списка веток используется команда

git branch

Получен следующий вывод

```
* dev01
  master
```

Символом * отмечена текущая ветка. Выполнение команды git stash list не отобразит информации, т. е. заначка, использованная для создания ветки, была удалена после использования. В файле 01.c появилась строка, спрятанная в заначку, а команда git status сообщает о наличии неучтенных изменений.

В случае наличия неучтенных изменений выполнить переключение на другую ветку не получится, т. к. из-за этого может случится некоторая путаница. Для предотвращения такой ситуации, необходимо, например, переключившись на ветку master командой

git checkout master

подтвердить переключение командой

git branch

В выводе символ * должен быть выведен напротив ветки master.

```
  dev01
* master
```

скрытые не выкаченные ветки
git branch —all

Далее, выполнив команду git status, будет получено сообщение о неучтенных изменениях в файле 01.c и в самом файле будет добавлена строка из заначки. Это происходит потому, что переключаясь на другую ветку, выполняется перемещение по указателям коммитов. Другим изменениям репозиторий не подвергается. Во избежания недоразумений, лучше вернуться на ветку dev01, командой

git checkout dev01

Отсутствие контроля над текущей веткой является одной из главных причин проблем с репозиториями у начинающих пользователей системы git.

Далее необходимо создать третий коммит в репозитории и первый в ветке dev01 с помощью команд

git add -A
git commit -m "dev01 first commit"

После этого необходимо запустить графическое приложение для работы с коммитами командой gitk. Отображаемая информация будет полностью соответствовать выполненным действиям.

Для **создания новой ветки** обычно используется не заначка, а коммит.

git branch <ИМЯ ВЕТКИ> <ИДЕНТИФИКАТОР КОММИТА>

Коммиты — это единицы, которыми могут обмениваться репозитории. К основным операциям по обмену коммитами относятся команды:

- git fetch — получить коммиты удаленного репозитория;
- git push — отправить коммит в удаленный репозиторий;
- git pull — комбинация команд получения и объединения коммитов.

Примеры использования приведенных команд для репозитория с названием по-умолчанию:

git push origin master
git pull origin master

В случае, если выполнение команды pull и включенной в неё команды автообъединения (automerge) завершилось ошибкой, будет выведено соответствующее сообщение. Сообщение содержит название команды, исправляющей полученные несоответствия. Также можно использовать команду

git mergetool

Система git имеет возможности тонкой настройки, которые сложны в освоении и используемые в основном руководителями проектов. Также присутствует ряд функций, освоение которых требует объемного описания, но они не относятся к возможностям первой необходимости. К ним можно отнести функцию перестроения базы коммитов git rebase или функцию получения произвольного коммита get chyrry-pick, git fast-forward, git reflog. Функция git rebase дописывает в качестве продолжения все коммиты одной ветки на другую ветку. Частный случай объединения веток git fast-forward, который применим только в случае отсутствия «наложения» изменений. Функция git reflog показывает

всю историю действий пользователя. Далее приведены команды, используемые в повседневной работе с репозиторием.

Для просмотра изменений по отношению к последнему коммиту используется команда

git diff filename

Способ сброса всех изменений до последнего коммита

git reset --hard HEAD

Для переключения на другой коммит с целью продолжения работы с него, обычно делают новую ветку

git checkout -b <НОВАЯ ВЕТКА> <ИДЕНТИФИКАТОР КОММИТА>

Опасный способ сброса изменений до последнего коммита, потенциально способный вызвать нарушение правильной работы коммитов

git reset --hard <ИДЕНТИФИКАТОР КОММИТА>

Команда отмены опубликованных коммитов, создавая на каждую отмену новый коммит

git revert <ИДЕНТИФИКАТОР КОММИТА1> <ИДЕНТИФИКАТОР КОММИТА2>

Добавление последних изменений к последнему локальному коммиту

git commit --amend

Для удаления локальной ветки используется команда

git branch -D <НАЗВАНИЕ ВЕТКИ>

Для удаления удаленной ветки

git push origin --delete <ИМЯ ВЕТКИ>

Сразу после инициализации репозитория необходимо задать некоторую глобальную конфигурационную информацию.

Имя пользователя имеет особую важность для репозитория. Выполняя создание коммитов, пользователь как бы подписывает каждую измененную строку. Git позволяет отображать автора изменений для каждой строки, что значительно упрощает совместную работу.

Для задания имени пользователя используется следующая команда

git config --global user.name "John Smit"

Также необходимо задавать адрес почтового ящика электронной почты, что значительно упрощает коммуникации во время совместной разработки

git config --global user.email "john@mail.com"

Для смены основных инструментов работы с исходными текстами программ используются команды

git config --global core.editor vim
git config --global merge.tool meld

Первая строка задает программу редактор файлов, вторая — средство для объединения разных версий файлов.

Для просмотра глобальных настроек используется команда

git config --list

В завершении раздела приведен список команд, необходимых для отправки файла в удаленный репозиторий, расположенный на популярном Интернет ресурсе http://githab.com.

git init
git add <ИМЯ ФАЙЛА>
git commit -m "first commit"
git remote add origin https://github.com/<РЕПОЗИТОРИЙ>/<ПРОЕКТ>
git push -u origin master

6.3. Этапы создания исполняемого файла. Типы исполняемых файлов

Исполняемый файл или файл программы — это файл с двоичной информацией в формате ELF. В каждой ОС принят свой формат исполняемого файла. Причем формат исполняемого файла зависит от архитектуры микропроцессора, поэтому в разных версиях одной ОС может отсутствовать переносимость исполняемых файлов.

Компилятор gcc может создавать как минимум 4 различных вариантов исполняемых файлов, используемых для разных целей:

- исполняемый файл по умолчанию (используется в процессе написания текста программы);
- исполняемый файл с отладочной информацией (используется для исправления логических ошибок в программе);
- исполняемый файл с информацией о производительности каждой функции программы (используется для оптимизации программы на уровне исходного текста программы);
- оптимизированный файл для внедрения и эксплуатации, с разной степенью оптимизации.

В приведенной последовательности они обычно и создаются.

В разделе 3.1 показан процесс создания исполняемого файла по классической схеме. Именно таким образом создается исполняемый файл компилятором gcc.

Рассмотрим создание исполняемого файла на примере файла main.c с исходным текстом программы.

Содержимое файла main.c

```
#include <stdio.h>
#define VALUE 3

int main () {
   printf("%d\n", VALUE);
   return 0;
}
```

Для выполнения первого этапа — обработки предпроцессором, выполняется команда

```
gcc -E main.c > main_E.c
```

В результате первая строка файла будет заменена содержимым файла stdio.h. Также будет замена константы VALUE на значение, указанное в директиве #define. В связи с тем, что результирующий файл содержит 856 строк, приведены только последние 7, включая пустые строки.

```
# 4 "main.c"
int main () {
   printf("%d\n", 3);
   return 0;
}
```

Следующим этапом выполняется компиляция полученного исходного текста программы без директив предпроцессора.

```
gcc -c main_E.c
```

В результате компиляции получен объектный файл main_E.o, в формате, определяемом конкретной версией компилятора.

Для получения исполняемого файла необходимо вызвать компилятор без ключей, указав ему входным файлом объектный файл.

gcc -o main_E.o -o main_E

В результате будет получен исполняемый файл, запускаемый из каталога, в котором он расположен командой

./main_E

В приведенных ранее примерах указывалось, при использовании математических функций во время компиляции необходимо указывать ключ -lm. Для указания этой библиотеки достаточно добавить этот ключ к указанной ранее строке.

gcc -o main_E.o -lm -o main_E

Без указания специального ключа программа собирается в режиме динамической сборки. В этом случае для запуска программы необходимо, чтобы файлы всех библиотек, используемых программой, были в наличии. При этом размер исполняемого файла минимален.

Выполнение статической сборки, когда все необходимые библиотеки добавляются в исполняемый файл, производится при использовании ключа -static.

gcc -o main_E.o -static -lm -o main_E

В этом случае, размер файла с программой может увеличится на два и более порядка.

Для просмотра всех настроек создания исполняемого файла и всех использованных для этого команд необходимо применить ключ -v.

gcc -v main.c

В полном выводе отображается вызов трех программ: ../configure, cc1 и collect2 с ключами и параметрами.

С целью вывода использованных при создании программы заголовочных файлов используется команда

gcc main.c --trace

Переданный аргумент trace будет передан компоновщику ld. В результате будет получен вывод, содержащий фактический набор использованных заголовочных файлов

. /usr/include/stdio.h
.. /usr/include/features.h
... /usr/include/sys/cdefs.h
.... /usr/include/bits/wordsize.h
... /usr/include/gnu/stubs.h
.... /usr/include/gnu/stubs-64.h
.. /usr/lib64/gcc/x86_64-alt-linux/5/include/stddef.h
.. /usr/include/bits/types.h
... /usr/include/bits/wordsize.h
... /usr/include/bits/typesizes.h
.. /usr/include/libio.h
... /usr/include/_G_config.h
.... /usr/lib64/gcc/x86_64-alt-linux/5/include/stddef.h

.... /usr/include/wchar.h
... /usr/lib64/gcc/x86_64-alt-linux/5/include/stdarg.h
.. /usr/include/bits/stdio_lim.h
.. /usr/include/bits/sys_errlist.h
. /usr/include/assert.h

Несколько защит подключения может быть полезно для:

/usr/include/assert.h
/usr/include/bits/stdio_lim.h
/usr/include/bits/sys_errlist.h
/usr/include/bits/typesizes.h
/usr/include/gnu/stubs-64.h
/usr/include/gnu/stubs.h
/usr/include/wchar.h

Для создания программы, соответствующей текущей версии стандарта языка программирования, используется ключ -std=c17. Без указания этого ключа синтаксические конструкции, утвержденные новым стандартом, будут помечены как ошибочные. При создании программ настоятельно рекомендуется избегать предупреждений от компилятора. Для этого их приравнивают к ошибкам с помощью ключей наиболее строгой проверки синтаксиса исходного текста программы (например, ключи -Wall и -pedantic). Кроме того, необходимо выполнять автоматическую оптимизацию исходных текстов программы с использованием ключей -O0, -O1, -O2, -O3.

Ключ -O3 использует наибольшее количество правил оптимизации исходных текстов, ключ -O0 соответствует отключению оптимизации. Следует иметь в виду, уровень оптимизации лучше повышать постепенно, проводя полнофункциональное тестирование всех функций. Неправильно выполненная оптимизация может изменить

логику работы программы или привести к ошибкам исполнения.

В результате команда создания исполняемого файла с максимальной оптимизацией из исходных текстов, соответствующих последнему стандарту языка программирования, должна выглядеть так:

gcc -Wall -pedantic -O3 -std=c17 main.c -o main

Исполняемый файл содержит только команды и константные значения. Для поиска ошибок в исходном тексте программы используется режим отладки. Этот режим отличается отсутствием оптимизации. В этом режиме исходный текст программы добавляется непосредственно в исполняемый файл, позволяя таким образом осуществить привязку двоичных команд к строкам исходного текста программы. Это еще одна причина, по которой каждое выражение исходного текста нужно писать в новой строке. Для включения отладочной информации используется ключ -g.

gcc -g main.c -o main

Файл с отладочной информацией имеет больший размер. Он может быть запущен как отдельная программа, которая выведет дополнительную информацию в случае её разрушения в процессе исполнения. Обычно, программы с отладочной информацией запускают с помощью программ отладчиков, например gdb. Основы работы с ней будут приведены в разделе 6.6 этой книги.

После исправления ошибок с использованием отладочной информации, выполняется проверка производительности разработанной программы. Сам

процесс называется профилированием программы. Для этого в исполняемый файл добавляются вызовы программы time, с помощью которой засекается время исполнения каждой функции.

Используя ЭТ и программу time, можно засечь время выполнения всей программы с помощью команды

time ./main

В результате будет выведено время, затраченное на исполнение программы, включая разбиение на время исполнения в режиме пользователя и режиме ядра ОС.

Профилирование программы выполняется в несколько этапов.

1. Создание исполняемого файла с информацией для профилирования с помощью команды

gcc -pg main.c -O3 -Wall -pedantic -std=c17 -o main

2. Исполнение программы, в процессе которого замеряется время исполнения функций и сохраняется в файл gmon.out

./main

3. выполняется преобразование в текстовый вид результатов, сохраненных в файле gmon.out с помощью программы gprof, используя команду

gprof ./01 > 1.txt

В приведенном примере результат преобразования сохраняется в текстовый файл 1.txt.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
72.07	0.05	0.05	1	50.45	50.45	calculate
28.83	0.07	0.02	1	20.18	20.18	calculate_sin
0.00	0.07	0.00	100000	0.00	0.00	my_sin

Далее идут пояснения

% time — the percentage of the total running time of the program used by this function.

cumulative seconds — a running sum of the number of seconds accounted for by this function and those listed above it.

self seconds — the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls — the number of times this function was invoked, if this function is profiled, else blank.

self ms/call — the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total ms/call — the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name — the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Затем граф вызовов

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 14.16% of 0.07 seconds

```
index % time    self  children    called     name
                                  <spontaneous>
[1]    100.0    0.00    0.07                 main [1]
               0.05    0.00       1/1          calculate [2]
               0.02    0.00       1/1          calculate_sin [3]
-----------------------------------------------
               0.05    0.00       1/1          main [1]
[2]     71.4    0.05    0.00         1        calculate [2]
-----------------------------------------------
               0.02    0.00       1/1          main [1]
[3]     28.6    0.02    0.00         1        calculate_sin [3]
               0.00    0.00  100000/100000      my_sin [4]
-----------------------------------------------
               0.00    0.00  100000/100000      calculate_sin [3]
[4]      0.0    0.00    0.00  100000         my_sin [4]
-----------------------------------------------
```

Повторный вызов привел к следующим результатам

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 67.27      0.02     0.02        1    20.18    20.18  calculate
 33.63      0.03     0.01        1    10.09    10.09  calculate_sin
  0.00      0.03     0.00   100000     0.00     0.00  my_sin
```

Все действия, необходимые для получения результатов профилирования, можно выполнить с помощью одной команды

gcc -pg main.c -O3 -Wall -pedantic -std=c17 -o main && ./main && gprof ./main > 1.txt

Во время профилирования следует выполнить программы многократно для получения средних значений, т. к. продолжительность работы функции может отличаться на порядок, особенно если функция исполняется быстро.

Кроме компилятора gcc существует множество

других компиляторов. На слабопроиводительных платформах также используются специальные компиляторы.

Для разработки программ с ядром ОС Линукс также может эффективно использоваться компилятор Clang. Он поддерживает почти все ключи, применяемые в gcc. Кроме того, он распространяется под лицензией, отличной от лицензии gcc, и позволяет создавать программы с закрытыми исходными текстами программы для коммерческих целей.

Для использования компилятора Clang в командах необходимо заменить только название компилятора следующим образом

clang -Wall -pedantic -O3 -std=c17 main.c -o main

Для проверки качества, надежности и переносимости разработанных исходных текстов программы их можно собирать разными компиляторами и проверять возможность сборки программы из исходных текстов и её работоспособность.

6.4. Автоматизация проверки исходных текстов и создания исполняемых файлов

Компилятор выполняет проверку только синтаксиса исходных текстов программы, практически не проверяя логические и прочие ошибки. Для поиска более сложных ошибок используются специальные автоматизированные средства проверки исходных текстов. Их принципиальное отличие в том, что компилятор для программиста друг, пытающийся понять как надо, а системы статического анализа исходных текстов программы действуют как враг, который указывает на все возможные и потенциальные ошибки. Для языка Си наиболее популярным средством исправления ошибок является программа **cppcheck**. Рассмотрим её применение с использованием следующего исходного текста программы (Рисунок 1).

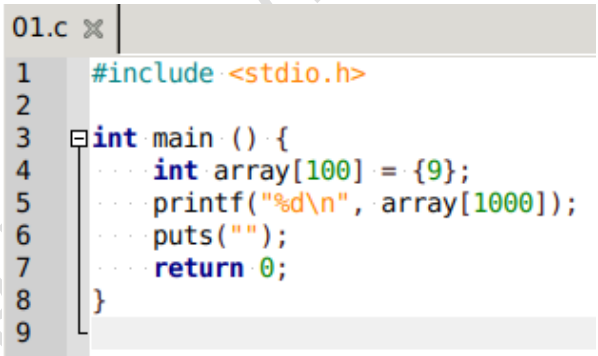


Рисунок 1 — Исходный текст для обработки cppcheck

Из исходного текста, представленного на рисунке 1, создается исполняемый файл без единого замечания даже в режиме максимально строгой проверки исходного текста.

gcc -pedantic -Wall -std=c17 01.c -o 01

Исполнение программы привело к выводу значения 1095517952. Ошибка заключается в том, что в строке 4 объявляется массив из 100 элементов, каждый из которых инициируется значением 9, однако в строке 5 производится обращение у несуществующему 1000-му элементу.

Вывод: ошибка в тексте программы не была выявлена ни на этапе создания программы, ни на этапе её исполнения.

Для выполнения проверки исходных текстов программы в текущем каталоге используется следующая команда

cppcheck --std=c11 .

На текущий момент актуальной являлась версия 1.82, поддерживающая только язык программирования Си стандарта 2011 года, что все равно позволяет применять эту утилиту для исходных текстов, соответствующих стандарту 2017 года. В результате было получено следующее сообщение

Checking 01.c ...
*[01.c:5]: (error) Array '*array[100]*' accessed at index 1000, which is out of bounds.*

Программа cppcheck правильно определила ошибку и место её нахождения. Не стоит ожидать, что программа также точно сможет определить все сложные ошибки, однако поможет исправить ряд досадных недоразумений, исправление которых потребует значительного времени.

Программа проверки имеет множество возможностей. В первую очередь следует ознакомиться с

возможностями:

- указывать целевую платформу, например – platform=win32A;
- запускать в многопоточном режиме, например, -j4 запускает в четыре потока;
- отображать большее количество информации, ключ -v;
- отображать ошибки в формате, совместимом с компилятором gcc, задается ключом --template=gcc.

Программа cppcheck может дополняться функциями и библиотеками, но в её дистрибутиве присутствует небольшая поддержка всего нескольких библиотек.

Важнейшим элементом любой современной интегрированной среды быстрой разработки программ является средство автоматизации компиляции программ. Для языка Си обычно это make, обрабатывающая конфигурационный скриптовый файл **Makefile**. Файл Makefile состоит из двух основных частей, списка правил и списка действий, например, собрать программу, установить программу, удалить программу;

Список правил позволяет собрать одну программу или несколько независимых программ. Каждое правило пишется как бы наоборот. Сначала указывается правило сборки конечного файла программы, а дальше детализируются команды сборки каждого из элементов, необходимого для сборки программы.

Синтаксис файла Makefile является достаточно строгим и не допускает замену символов — разделителей, их количества и прочих вольностей. В начале правила указывается имя результирующего файла, называемое «цель». Затем ставится двоеточие. Далее через пробел указываются «зависимости», т. е. полный список файлов, необходимых для сборки цели. Обычно зависимости также создаются в процессе

исполнения Makefile, поэтому для создания файла, входящего в зависимости, используются команды. В одной строке должна быть строго одна команда, которая должна быть отделена от начала строки одним символом горизонтальной табуляции. При этом правила Makefile могут содержать переменные, конструкции ветвления и циклы.

Для демонстрации применения утилиты make используем исходный текст программы из пункта 3.7 этой книги, который отображен на рисунке 3 (файл 03_1.c), рисунке 4 (файл 03_2.h) и рисунке 5 (файл 03_2.c). В том же каталоге создан Makefile следующего содержания:

```
test_make: 03_1.o 03_2.o
	gcc 03_2.o 03_1.o -o test_make
03_1.o: 03_1.c
	gcc -pedantic -Wall --std=c17 -c 03_1.c
03_2.o: 03_2.h 03_2.c
	gcc -pedantic -Wall --std=c17 -c 03_2.c
clear:
	rm -f test_make 03_1.o 03_2.o
install:
	cp test_make ../
uninstall:
	rm -f ../test_make
```

Одним из наиболее очевидных удобств, предоставляемых Makefile, кроме отсутствия необходимости каждый раз писать длинные команды, является возможность многопоточного запуска компиляции.

В Makefile используется две цели. Для их параллельной сборки на двух разных ядрах необходимо

выполнить следующую команду

make -j2

При наличии множества целей, можно указывать количество потоков, равное количеству аппаратных ядер центрального процессора. Причем, если микропроцессор имеет 4 аппаратных ядра, в каждом из которых может эмулироваться по два потока, всего 8, наибольшая производительность будет достигнута при указании количества потоков, равное количеству аппаратных ядер, т. е. 4.

Для установки программы необходимо выполнить команду

make install

В приводимом примере установка заключается в копировании исполняемого файла на каталог выше.

Для удаления вспомогательных файлов, создаваемых в процессе сборки программы, необходимо выполнить команду

make clear

В данном примере будут удалены не только файлы библиотек, но и исполняемый файл программы. Утилита make не регламентирует жестко названия команд. Общепринятой практикой является использование сочетания make clear, однако в примере последняя буква заменена на n, что никак не сказалось на исполнении программы.

Удаление файла программы и всех прочих элементов программы выполняется командой

make uninstall

В примере этой командой удаляется файл, находящийся на уровень выше в месте установки.

Использование программы make позволяет упростить ряд операций по созданию программы, однако не обеспечивает легкую переносимость между ОС разных классов. Для решения этой проблемы используется пакет GNU Autotools. Обобщенный алгоритм использования этого средства автоматизации заключается в запуске программ autoscan, autoconf с дальнейшим заданием конфигурационной информации в автоматически сгенерированных файлах configure.ac и Makefile.am. Для создания Makefile используется команда ./configure.

Далее рассматривается порядок действий для автоматического получения Makefile. Перед началом выполнения последовательности действий необходимо подготовить каталог с подкаталогом src. В каталоге src должны размещаться файлы с исходным текстом программ. Представим вывод команды tree корневого каталога проекта с правильно подготовленными каталогом и файлами

```
.
|--src
|  |--03_1.c
|  |--03_2.h
|  |--03_2.c
|--configure.ac
|--Makefile.am
```

Команда tree в дистрибутиве АльтЛинукс не устанавливается при установке ОС. Её необходимо

устанавливать дополнительно из репозитория.

1. Изменить содержимое файла configure.ac на

AC_INIT(src/03_1.c)
AM_INIT_AUTOMAKE(test_make,0.1)
AC_PROG_CC
AC_PROG_CXX
AC_PROG_INSTALL
AC_OUTPUT(Makefile)

Заготовку файла можно получить, выполнив команду autoscan, в корневом каталоге проекта. В результате будет получен файл configure.scan. Он переименовывается в configure.ac, но его содержимое будет отлично от предлагаемого.

2. Изменить содержимое файла Makefile.am на

bin_PROGRAMS=test_make
test_make_SOURCES=src/03_2.h src/03_2.c src/03_1.c

Пояснения к командам, добавленным в измененные файлы, доступны в сети интернет. Их освоение не составит труда.

3. Выполнить следующие команды

aclocal
autoconf
touch README AUTHORS NEWS ChangeLog
automake -a
./configure
make

Программа aclocal выполняет подготовительную

работу, необходимую для генерации скрипта configure и Makefile. Программа autoconf формирует скрипт configure. Программа touch создает файлы с именами, соответствующими переданным в неё параметрам. Программа automake генерирует файл Makefile.in, на основе которого формируется Makefile. Скрипт configure располагается в корневом каталоге проекта. Его выполнение приводит к генерации Makefile. Используя Makefile, утилита make генерирует файл программы.

При подготовке пакета для распространения вместо последнего вызова make необходимо выполнить

make dist

В результате будет собран архив test_make-0.1.tar.gz. Для установки пакета, собранного командой make dist, необходимо выполнить следующие действия

tar -xvvf test_make-0.1.tar.gz
cd test_make-0.1
./configure
make
make install
make clean
test_make

Для удаления установленной программы используется команда

make uninstall

В случае, если проект состоит из пары десятком файлов, более простым выглядит создание Makefile в текстовом редакторе с целью экономии времени на

сборку проекта. Для работы с крупными кроссплатформенными проектами предпочтительнее использовать возможности, предоставляемые средством GNU Autotools.

6.5 Автоматическая генерация документации

Документирование исходных текстов программы требует значительных затрат времени и может приводит к задержке сдачи проекта. Основным средством автоматизации процесса создания документации является программа doxygen.

В соответствии с жизненным циклом программ и этапов их разработки, написание комментариев происходит до написания текста программы. Для эффективного использования doxygen достаточно начинать каждый файл с исходным текстом программы с комментарием в формате doxygen, также необходимо комментировать составные типы данных и функции.

Приведем пример с использованием файлов с исходным текстом программы из пункта 3.7 этой книги, который отображен на рисунке 3 (файл 03_1.c), рисунке 4 (файл 03_2.h) и рисунке 5 (файл 03_2.c).

В начало каждого файла добавлен блок комментария по примеру следующего фрагмента

```
/*!
\file
\brief Файл содержит точку входа в программу

Данный файл содержит в себе точку входа в программу
*/
```

Перед каждой функцией добавлен фрагмент по следующему шаблону

```
/**
 * \brief Точка входа
```

```
 *
 * Исполнение программы
 * начинается здесь.
 *
 * \param argc Количество аргументов
 * \param argv Список аргументов
 *
 * \return Статус завершения программы
 */
```

Использованными ключевыми словами являются следующие сочетания символов:

\file – добавляется в начале каждого файла
\brief – краткое описание файла/блока/функции
\param – параметр, передаваемый в функцию
\return – возвращаемое значение
После блока \brief с отступом одна пустая строка выполняется более подробное описание элемента.

Программа doxygen использует множество других ключевых слов, наиболее часто используемыми из которых являются следующие:

\authors – указание авторов
\version – содержит версию фрагмента текста программы
\date – позволяет указывать дату создания текста программы
\bug – используется для перечисления известных ошибок
\warning – предупреждения для программистов
\copyright – лицензионные ограничения, действующая лицензия

\example – содержит ссылку на пример использования
\todo – содержит список запланированных работ

Для описания выражения после него используется конструкция

double value; ///< Краткое описание

Для описания составных типов может использоваться следующая конструкция

```
/// Состояния объекта
enum States {
    Undefined ///< Элемент в неопределенном состоянии
    Enabled, ///< Элемент активирован
    Disabled, ///< Элемент отключен
}
```

или

```
/// Элемент
struct Items {
    int on; ///< Элемент активен или нет
    double value; ///< Значение элемента
} item;
```

После оформления комментариев указанным ранее образом, выполняется команда создания проекта doxygen

doxygen -g

В результате создается файл проекта с именем по

умолчанию Doxygen. Для создания файлов с описанием проекта выполняется вызов программы без аргументов.

doxygen

В результате будут созданы каталоги html и latex. Для просмотра созданной документации, необходимо начать просмотр файла index.html из каталога html. На рисунке 1 приведен пример автоматически сгенерированного описания функции main().

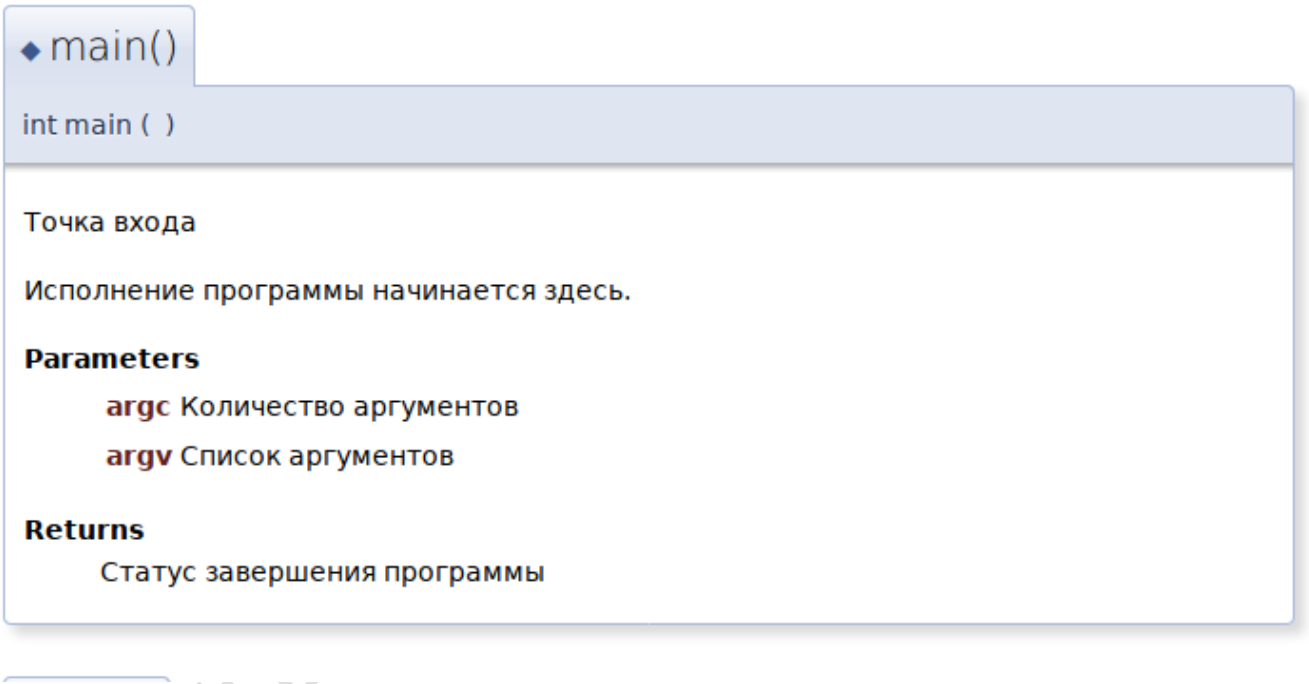


Рисунок 1 — Пример описания функции main

Программа doxygen имеет множество настроек, которые можно изменять как с помощью текстового редактора, так и используя графический интерфейс. Для запуска графического интерфейса конфигуратора применяется команда doxygenwizard .

6.6. Приемы отладки программ

Рассмотрение приемов поиска ошибок в исходных текстах программ (отладки программы) будет произведено на примере исходного текста, представленного на рисунке 1.

01.c

```
1   #include <stdio.h>
2
3   int max_value (int first, int second) {
4       fprintf(stderr, "Функция запускается\n");
5       int value = first > second ? first : second;
6       fprintf(stderr, "Функция выполнена\n");
7       return value;
8   }
9
10  int main () {
11      int * first;
12      int second = 3;
13      perror("Вызов функции");
14      int value = max_value (*first, second);
15      perror("Вызов прошел успешно");
16      printf ("Максимальное значение: %d\n", value);
17      return 0;
18  }
19
```

Рисунок 1 – Исходный текст программы с ошибкой

В приведенном исходном тексте реализовано две функции max_value() и main(). В функции max_value() в строках 4 и 6 использованы функции вывода fprintf(), первым аргументом которых указан стандартный поток ошибок stderr. Соответственно весь их вывод будет помещен в стандартный поток ошибок.

В функции main() в строках 13 и 15 вызывается функция perror(), объявленная в файле stdio.h. Функция perror() выводит текстовый аргумент и затем текстовое описание числового кода последней ошибки в стандартный поток ошибок.

Пояснение для остальных строк будут даны по мере демонстрации способов и технологий поиска ошибок.

В случае сборки исполняемого файла и его запуска командой

gcc 01.c && a.out

будет получен вывод

Вызов функции: Success
Ошибка сегментирования

В соответствии с выводом, после запуска программы её сегментация была разрушена и она уничтожена. Имея понимание синтаксиса языка Си, можно достаточно легко найти ошибку, однако это будет сделано с использованием разных приемом в технологии в разделах 6.6.1, 6.6.2 и 6.6.3.

6.6.1 Отладка без специальных программ

Выполнять поиск ошибок в режиме исполнения программы допустимо без специальных средств, упрощающих эту работу. В этом пункте речь идет об ошибках режима исполнения, процесс исправления которых часто называется отладкой. Наиболее сложным способом поиска и исправления ошибок пользовалась Ада Левис, первый программист на Земле. Всю работу по поиску и исправлению ошибок ей приходилось производить в уме. Некоторые разработчики наивысшей квалификации умеют находить ошибки исключительно путем чтения исходных текстов программы. Такой подход свойственен более старшим разработчикам, некоторые из которых застали время, когда написание исходных текстов программ велось с помощью карандаша и листа бумаги. Набор текста программы производился только после его написания на бумаге.

Необходимость в навыке поиска ошибок режима исполнения программы без специальных программ возникает в случае отсутствия средств отладки для некоторой малораспространенной или новой платформы.

Основными задачами поиска ошибок в программе являются поиск места, в котором произошла ошибка, и значения, приведшего к ошибке. Используя средства ОС и средства языка программирования, применяются следующие приемы для поиска места ошибки и значения, которое привело к ошибке:

1) стандартные функции вывода, например puts(), printf();

2) специальные версии операторов вывода, например perror();

3) перенаправление стандартного потока ошибок в файл (пункт 5.7 данной книги, рисунок, строка 13);

4) подключаемую библиотеку assert.

5) директивы предпроцессора.

Первый и второй способы продемонстрированы в исходном тексте в пункте 6.6. В этом же пункте показан вывод, производимый программой при её запуске. Первая строка вывода содержит сообщение

Вызов функции: Success

Вывод сгенерирован строкой 13 файла 01.c. Вторая строка сообщает о разрушении программы в ОЗУ. Вывод в строке 15 сообщения "Вызов прошел успешно" не выполнен, из этого делается вывод, что строка 14 не была выполнена.

В строке 14 производится вызов функции max_value(). Функция начинается с вызова оператора вывода в строке 4, который выводит сообщение в стандартный поток ошибок. Делается вывод, функция не начала выполняться.

Определено, что ошибка произошла на строке вызова функции, причем сама функция выполняться не начала. Значит ошибка заключается в неправильном задании аргументов функции, что и приводит к разрушению программы.

Функция max_value() принимает два аргумента целочисленного типа int, причем к первому аргументу применяется унарная операция разыменования, обозначаемая символом "*". В строке 11 выполняется объявление переменной first как указателя на целочисленное значение, однако переменная не инициализируется и может иметь «мусорное» значение. При выполнении проверки аргументов во время вызова функции max_value() происходит разрушение памяти программы.

Ошибка найдена. Она в строке 11. Чтобы её исправить, необходимо заменить строку

*int * first;*

например, на

int a = 2;
*int * first = &a;*

Приведенный метод отладки с использованием операторов вывода является самым трудоемким и требует повышенной внимательности. Рекомендуется применять его только в крайних случаях.

Для проверки правильности внесенных исправлений необходимо выполнить пересборку программы, запустив её заново командой

gcc 01.c && a.out

будут выведены следующие сообщения

Вызов функции: Success
Функция запускается
Функция выполнена
Вызов прошел успешно: Success
Максимальное значение: 3

Следует обратить внимание на то, что сообщения, выводимые в стандартный поток ошибок, также выведены в стандартный поток вывода. Следует отметить, сообщения выведены в порядке исполнения строк по исходному тексту программы, поэтому еще одной причиной использования такого приема является

необходимость проверки порядка исполнения строк текста программы в разных функциях или модулях.

В случаях, когда отладочный вывод достигает сотен и тысяч строк, рекомендуется перенаправлять поток ошибок в файл. Если ошибки появляются не более раза в несколько секунд, возможно использование приема, продемонстрированного в конце пункта 5.7 этой книги с тем отличием, что перенаправление производится на стандартный поток ошибок с кодом 2.

Для демонстрации работы перенаправления необходимо в одном ЭТ выполнить команду tty и получить примерно такой вывод

/dev/pts/0

В другом терминале запустить программу обязательно с исправленной, как указано ранее, ошибкой с помощью команды

./a.out 2>/dev/pts/0

В результате в ЭТ, в котором произведен вызов программы, будет выведена всего одна строка

Максимальное значение: 3

В терминале /dev/pts/0 будут выведены следующие сообщения

Вызов функции: Success
Функция запускается
Функция выполнена
Вызов прошел успешно: Invalid argument

Прием с двумя ЭТ позволяет очистить стандартный поток вывода от отладочных сообщений, позволяет сосредоточиться на выводе приложения или поиске ошибки с помощью отладочных сообщений.

Внесение изменений в исходный текст программы с целью поиска ошибок требует его восстановления к исходному виду после её нахождения. Однако и в этом случае в языке Си есть специальный механизм, позволяющий не изменять текст программы и собирать её в нескольких вариантах с помощью директив предпроцессора или макросов assert().

Для демонстрации четвертого способа поиска ошибок используем следующий исходный текст программы

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

int main (void) {
   FILE *fd;
   fd = fopen ("/home/user/file.txt", "r");
   assert (fd);
   fclose (fd);
   return 0;
}
```

При запуске программы, полученной из этого исходного текста, будут выведены следующие сообщения

```
a.out: 02.c:7: main: Assertion `fd' failed.
Аварийный останов
```

Механизм работы библиотеки assert описан в разделе 7.2 стандарта языка программирования Си. При

использовании этой библиотеки по-умолчанию, компилятор включает её и все макросы assert() в тексте программы. Для их отключения необходимо использовать следующие ключи компилятора -D NDEBUG. В этом случае команда сборки программы будет выглядеть следующим образом

gcc 02.c -D NDEBUG

После запуска программы собранной с отключенными макросами assert(), будет получено одно сообщение

Ошибка сегментирования

Программа разрушилась в процессе выполнения. Такое поведение связано с отсутствием открываемого файла /home/user/file.txt с правами на чтение. В этом случае функция fopen() не создала структуру типа FILE и при вызове функции fclose() в ней в качестве аргумента был передан указатель на структуру FILE, под которую не было выделено памяти в ОЗУ, что и привело к разрушению программы.

Наиболее гибким инструментом для поиска ошибок является комбинация директив предпроцессора и операторов вывода. Такой прием позволяет произвольным образом изменять функциональность программы в отладочном режиме, загружать или не загружать любые библиотеки, выполнять произвольную работу с любыми ресурсами. Главный недостаток этого подхода — в необходимости выполнять проектирование и разработку программы для двух режимов работы (отладочный и режим эксплуатации).

03.c

```c
#include <stdio.h>

//#define DEBUG

int max_value (int first, int second) {
    #ifdef DEBUG
    fprintf(stderr, "Функция запускается\n");
    #endif
    int value = first > second ? first : second;
    #ifdef DEBUG
    fprintf(stderr, "Функция выполнена\n");
    #endif
    return value;
}

int main () {
    int a = 2;
    int * first = &a;
    int second = 3;
    int value = max_value (*first, second);
    printf ("Максимальное значение: %d\n", value);
    return 0;
}
```

Рисунок 1 — Исходный текст программы с директивами предпроцессора, используемыми для отладки программы

Переключение в режим отладки возможен двумя способами:

- использование директивы предпроцессора с указанием выбранного ключевого слова;
- использование ключа -D при запуске программы.

На рисунке 1 представлен пример исходного текста программы из вступления к пункту 6.6. В тексте исправлена ошибка и он дополнен директивами предпроцессора, позволяющими продемонстрировать переключение в отладочный режим и обратно.

Далее описаны отличия исходного текста программы файла 03.c от исходного текста, описанного в пункте 6.6.

В строке 3 директива предпроцессора закомментирована. Удаление символов комментирования приведет к объявлению значения DEBUG, используемого предпроцессором. Объявление значения с таким идентификатором приведет к включению в компилируемый текст программы участков исходного текста, находящихся между директивами

#ifdef DEBUG

и ближайшим

#endif

В случае, если строка 3 закомментирована, текст между этими директивами включен не будет.

Второй способ переключения в режим отладки заключается в вызове сборки программы следующим образом:

gcc 03.c -D DEBUG

или так

gcc 03.c -D DEBUG=1

Значение константы DEBUG не имеет значения. Запуск полученной программы выполняется следующим образом

./a.out

Если программа собрана любым из описанных способов без отладочной информации, после её запуска

будет выведено одно сообщение

Максимальное значение: 3

Добавление отладочной информации любым способом приведет к выводу отладочной информации, путем срабатывания функций в строках 7 и 11 на рисунке 1.

Функция запускается
Функция выполнена
Максимальное значение: 3

Навык поиска ошибок режима исполнения без специальных программ будет всегда востребован на переднем крае технологической гонки. В таких областях всегда сначала появляются новые информационные технологии, а полноценное информационное и технологическое обеспечение к ним становится доступным не ранее чем через несколько лет успешных внедрений. К тому времени, когда полноценные средства отладки добираются в новую область, разработка с их помощью обесценивается также, как и стоимость работ, проводимых с их помощью, а значит и доходов разработчиков.

6.6.2. Отладка с использованием gdb

Наиболее популярным отладчиком в ОС Линукс является gdb. Для его полноценного изучения работе с ним необходимо посвятить целую книгу, поэтому в этом разделе будут даны наиболее часто используемые команды, приведен простой пример использования gdb и описана программа с графическим интерфейсом nemiver, позволяющим выполнять отладку, постоянно держа перед глазами значения регистров, ячеек ОЗУ и прочую информацию.

Отладчик gdb упрощает и ускоряет поиск ошибок, допущенных программистом и пропущенных компилятором и компоновщиком. Он позволяет контролировать выполнение другой программы, выполнять её построчно, в процессе выполнения просматривать и изменять произвольные значения в ОЗУ, перемещаться по стеку. Для работы с ним чаще других используются следующие команды:

file <ИМЯ ФАЙЛА> - загрузка в отладчик файла программы;

gdb <ИМЯ ФАЙЛА> - запуск отладчика из ЭТ с одновременной загрузкой исполняемого файла для отладки;

run или **r** – запуск исполняемого файла;

kill – завершение процесса отладки;

list main.c:3,10 – отобразить исходный текст программы из файла main.c с 3 по 10 строки;

break main.c:15 – установить точку останова в строке 15 файла main.c;

break 15 – установить точку останова в строке 15 текущего файла;

delete 1 или **d 1** – удаление точки останова с номером 1;

clear – удалить все точки останова в текущей функции;

delete или **d** – удалить все точки останова;

info breakpoints – просмотр существующих точек останова;

print variable или **p variable** – вывод значения переменной в текущей точке останова, если переменная является массивом выводятся значения всех элементов массива;

p * array@len – вывод всех значений элементов массива с идентификатором array, объявленного следующим образом int *array = (int *) malloc (len * sizeof (int));

display variable – вывод на экран значения переменной в каждой точке останова;

watch variable – останавливать выполнение программы при изменении переменной, считается контрольным выражением;

info watchpoints – просмотр всех контрольных выражений;

frame – отображение текущего кадра, при передачи адреса кадра или его номера выводится содержимое выбранного кадра;

continue или **c** – продолжить выполнение программы после точки останова;

set variable=0 – задание значения 0 для переменной с идентификатором variable в текущей точке останова;

step или **s** – пошаговое выполнение после текущей точки останова;

next или **n** – пошаговое выполнение в пределах функции, в случае вызова функции переход в неё не осуществляется;

finish – прекращение пошагового исполнения;

backtrace или **bt** – показывает список вызовов

функций в точке останова;

where – просмотр содержимого стека в точке останова;

up – подняться вверх на одну функцию по стеку вызовов, используется для просмотра значений переменных и констант в вызвавшей функции;

down – опуститься по стеку вызовов на одну функцию вниз;

set follow-fork-mode <parent/child> – указывает направление перехода при создании нового процесса функцией fork(); parent – оставаться в процессе родителе, child – идти в процесс потомок;

info registers – вывести имена и содержимое всех регистров в текущем кадре, кроме регистров с плавающей запятой;

info all-registers – вывести имена и содержимое всех регистров;

info registers <ИМЯ РЕГИСТРА> - список имен регистров, которые необходимо вывести;

kill – завершение отладки программы;

quit или **q** – выход из отладчика;

info float – отобразить аппаратно-зависимую информацию о модуле поддержки вычислений с плавающей запятой;

apropos local – поиск в подсказке по слову local в процессе работы с отладчиком gdb;

info local – поиск в подсказке пояснений к команде local в процессе работы с отладчиком gdb;

gdb info locals – поиск в подсказке пояснений к команде local из ЭТ.

В процессе описания команд были использованы следующие базовые понятия, характерные для всех отладчиков:

точка останова – остановка исполнения процесса под отладкой и получение возможности просматривать и изменять значения в ОЗУ;

контрольное выражение – условие, при котором выполняется некоторое действие, обычно остановка исполнения программы;

кадр – область видимости, обычно значения в функции и значения регистров при переключении в функцию для её исполнения.

Отладка с помощью gdb возможна для любых исполняемых файлов. Отладка файлов без отладочной информации делает недоступными ряд ключевых функций, ускоряющих отладку. Для включения отладочной информации в исполняемый файл необходимо при его компиляции указывать ключ -g.

gcc 01.c -g

При использовании Makefile передать ключ -g для его использования при компиляции каждого файла можно в коммандной строке ЭТ следующим образом:

make CFLAGS=-g

Или добавив в Makefile следующую строку

CFLAGS=-g

Для запуска gdb использована команда

gdb a.out

После этого возможен просмотр содержимого

исходного текста текущего файла

```
(gdb) list 1,9
1    #include <stdio.h>
2
3    int max_value (int first, int second) {
4        fprintf(stderr, "Функция запускается\n");
5        int value = first > second ? first : second;
6        fprintf(stderr, "Функция выполнена\n");
7        return value;
8    }
9
```

Для установки точки останова в строке 5 используется команда

```
break 5
```

Будет выведено следующее сообщение

```
Breakpoint 1 at 0x400622: file 01.c, line 5.
```

Производится запуск программы на исполнение

```
r
```

Получено следующее сообщение

```
Starting program: /home/a.out
Missing separate debuginfo for /lib64/ld-linux-x86-64.so.2
Try to install the hash file /usr/lib/debug/.build-id/6c/b5953b40bba0b1f74202ae673850709ec920d4.debug
Missing separate debuginfo for /lib64/libc.so.6
```

Try to install the hash file /usr/lib/debug/.build-id/14/a293becbf38dc6c9e8779938147ec7c0c51f15.debug
Вызов функции: Success

Program received signal SIGSEGV, Segmentation fault.
0x0000000000040066f in main () at 01.c:14
*14 int value = max_value (*first, second);*

Отладчик указал на тип ошибки и строку, которая стала её источником.

Несмотря на то, что программа разрушена, в памяти присутствует возможность просмотра значений аргументов в точке разрушения. Далее приведены команды вывода значений всех переменных, использованных в строке 14.

*(gdb) print *first*
Cannot access memory at address 0x0
(gdb) print second
$1 = 3
(gdb) print value
$2 = 32767

В соответствии с выводом, переменная *first указывает в значение по адресу 0x0, что и привело к ошибке. Значение second равно 3. Значение value имеет мусорное значение 32767.

Для проверки стека вызовов использована следующая команда

bt

Стек вызовов содержит вызов только одной функции

#0 0x00000000040066f in main () at 01.c:14

Для вывода значений регистров использована команда

info registers

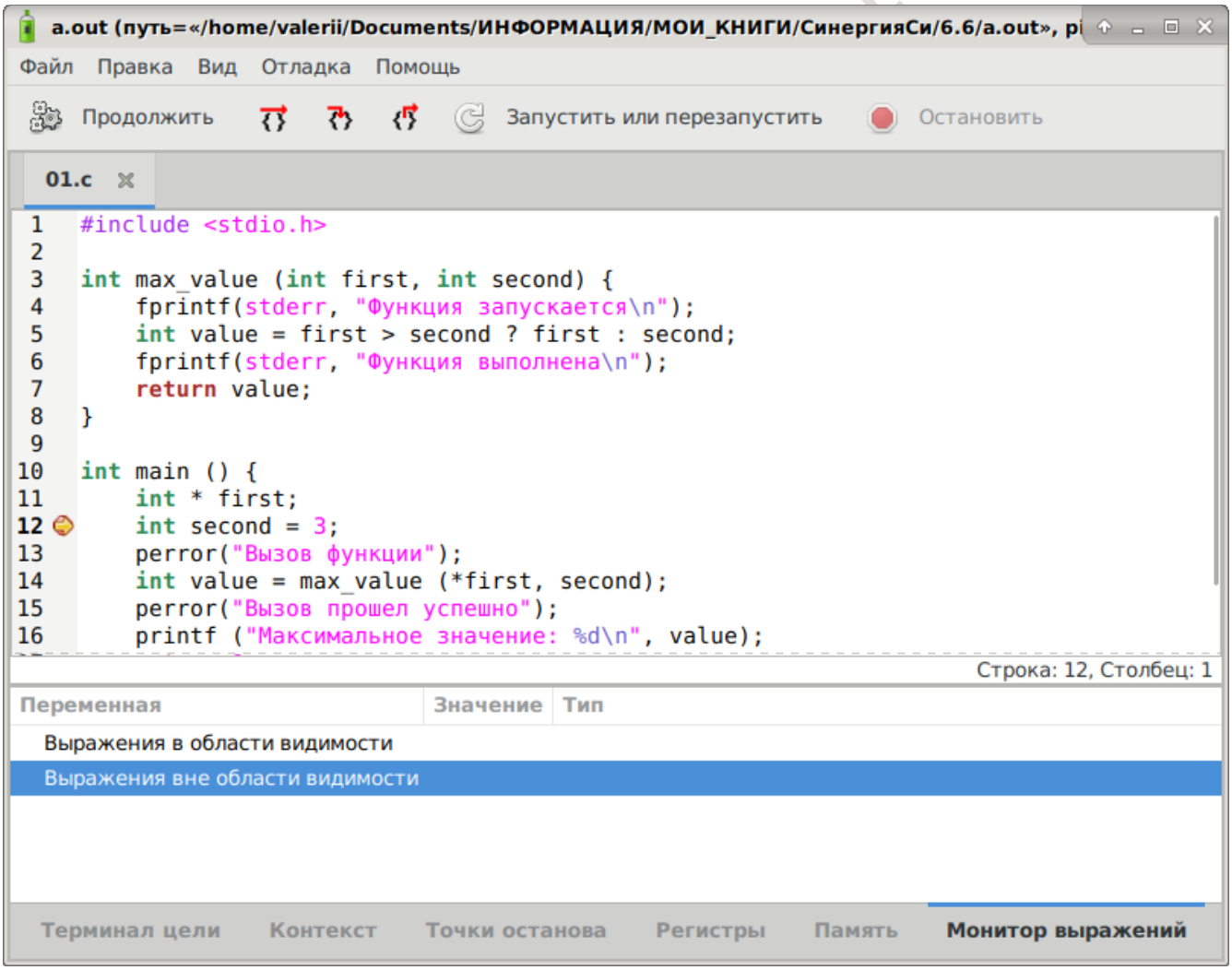
Отображены следующие значения регистров.

```
rax            0x0     0
rbx            0x0     0
rcx            0x7ffff7ffcc80   140737354124416
rdx            0x7ffff7dd4380   140737351861120
rsi            0x7ffff7dd3b58   140737351859032
rdi            0x20000    131072
rbp            0x7fffffffe420   0x7fffffffe420
rsp            0x7fffffffe410   0x7fffffffe410
r8             0x7ffff7fd0700   140737353942784
r9             0x23    35
r10            0x7     7
r11            0x246   582
r12            0x400500   4195584
r13            0x7fffffffe500   140737488348416
r14            0x0     0
r15            0x0     0
rip            0x40066f   0x40066f <main+29>
eflags           0x10206   [ PF IF RF ]
cs             0x33    51
ss             0x2b    43
ds             0x0     0
es             0x0     0
fs             0x0     0
gs             0x0     0
```

Для завершения отладки выполняется команда **q** и сразу же завершение подтверждается нажатием **y**.

Преимущество консольного отладчика – в возможности выбора графической оболочки для работы с ним. Для примера выбрана оболочка nemiver. Программа имеет полноценный оконный интерфейс. Её можно запустить как с помощью выбора в меню графического интерфейса или с помощью команды

nemiver <ИМЯ ПРОГРАММЫ> <АРГУМЕНТ 1> ... <АРГУМЕНТ N>



6.6.3. Поиск утечек памяти

Применение приемов и технологий исправления ошибок режима исполнения не всегда способна выявить ошибки, возникающие при работе с памятью. Некорректное освобождение ресурсов может порождать скрытые ошибки, проявляющиеся только на одной из платформ.

Наиболее часто встречающиеся ошибки объединены одним названием – утечка памяти. К этому понятию относится выделение памяти ОЗУ со стороны ОС для программы и её не освобождение. Другим источником ошибок является обращение в область памяти, которая уже возвращена в ОЗУ или никогда не выделялась.

Для проверки корректности работы с памятью ОЗУ используются специальные средства, самым популярным из которых является набор инструментов valgrind. Наиболее часто используется программа memcheck. Она может применяться как к программам, в которые включена отладочная информация, так и к тем, в которые отладочная информация не включена.

Проверка с помощью утилит поиска утечек памяти обычно производится после отладки программы. Иногда приходится её выполнять в процессе написания и отладки при возникновении ошибок, причину которых выяснить не получается другими способами.

Демонстрация поиска утечек памяти в ОЗУ будет проводиться на примере исходного текста программы из пункта 6.6. Для этого необходимо выполнить сборку программы с использованием ключа -g

gcc 01.c -g

Запуск программы выполняется следующим образом

valgrind --leak-check=full ./a.out

Программа добавляет в стандартный поток вывода очень подробное описание. Не все найденные ошибки являются ошибками, возникшими по причине неверно написанного исходного текста программы. Другими причинами ошибок могут стать особенности платформы, отличия в версия ОС или компилятора. Также иногда случаются ложные срабатывания, но на практике такое происходит в самых редких случаях.

Наиболее важными строками вывода являются:

==2455== Command: ./a.out
==2455==
Вызов функции: Success
==2455== Use of uninitialised value of size 8
==2455== at 0x40066F: main (01.c:14)
==2455==
==2455== Invalid read of size 4
==2455== at 0x40066F: main (01.c:14)
==2455== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==2455==
==2455==
==2455== Process terminating with default action of signal 11 (SIGSEGV)
==2455== Access not within mapped region at address 0x0
==2455== at 0x40066F: main (01.c:14)
...
==2455== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
Ошибка сегментирования

В конце вывода содержится строка ERROR SUMMARY, в которой указывается общее количество ошибок. В случае отсутствия ошибок читать длинный вывод не имеет смысла.

В выводе инструмента поиска ошибок утечки памяти точно указана строка, имя файла с исходным текстом программы, которая привела к ошибке main (01.c:14), а также название функции и адрес в памяти ОЗУ.

В случае, когда программа не содержит отладочной информации, в выводе будет указано только название функции и адрес в ОЗУ, а также причина ошибки. При этом количество ошибок будет также указано правильно.

==2918== Command: ./a.out
==2918==
Вызов функции: Success
==2918== Use of uninitialised value of size 8
==2918==   at 0x40066F: main (in /home/a.out)
==2918==
==2918== Invalid read of size 4
==2918==   at 0x40066F: main (in /home/a.out)
==2918== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==2918==
==2918==
==2918== Process terminating with default action of signal 11 (SIGSEGV)
==2918== Access not within mapped region at address 0x0
==2918==   at 0x40066F: main (in /home/a.out)
...
==2918== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

Ошибка сегментирования

Программа valgrind имеет множество других возможностей отладки и профилирования исходного текста программы и исполняемого файла, однако описанный способ позволяет приступить к её использованию без детального изучения её возможностей.

6.7. Си и ассемблер. Технологии дезасемблирования

Язык программирования Си вырос из информационных технологий, разрабатывавшихся на языке ассемблер. Кроме того, он изначально использовался для разработки ОС, часть которой всегда написана на ассемблере. В связи с этим язык Си легко интегрируется с модулями, разработанными на ассемблере. В данном разделе не будет исчерпывающих пояснений исходного текста на языке ассемблер. Его изучение требует наличие знаний в области микропроцессорных технологий, аппаратного обеспечения ЭВМ и как минимум основ построения ОС.

Рассмотрим пример использования функции pow, написанной на языке ассемблер и размещенной в файле pow.s.

pow.s

```
.globl pow
pow:
  movl -4(%rbp), %eax
  imull %eax, %eax # умножить само на себя
  ret
```

Для использования функции pow использован следующий текст программы на языке Си, размещенный в файле main.c .

main.c

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

```
int pow (int); /*На ассемблере*/

int main () {
  int i = 4;
  printf("Before pow i = %d\n",i);
  i = pow(i);
  printf("After pow i = %d\n",i);
  return 0;
}
```

Компилятор gcc поддерживает работу с текстом на языке ассемблер двух нотаций intel и AT&T. В первом случае предложено скомпилировать сначала объектный файл из текста на языке ассемблер, затем выполнить сборку исполняемого файла.

```
gcc -c pow.s
gcc pow.o main.c
```

Для компилятора gcc ассемблер является поддерживаемым языком, поэтому нет необходимости разделять этапы сборки исполняемого файла. Она выполняется одной командой так, как если бы использовались файлы, написанные только на языке Си.

```
gcc pow.s main.c
```

После сборки программы её запуск осуществляется как обычно

```
./a.out
```

В результате выполнения программы будет получен следующий вывод

Before pow i = 4
After pow i = 16

Использование исходных текстов на языке Си допускается в программах на языке ассемблер. Далее приведен пример программы на ассемблер, использующую функцию на языке Си. Файл print.c содержит следующий исходный текст

print.c

```
# include <stdio.h>

void print(int i) {
  printf("%d\n",i);
}
```

Текст главной функции программы реализован в файле main.s и написан на языке ассемблер.

main.s

```
.globl main
main:
  pushq  %rbp # Кладем в стек адрес возврата
  movq   %rsp, %rbp # Сохраняем вершину стека
  movl $1123, -4(%rbp) # Кладем в стек за вершиной стека первую константу
  movl -4(%rbp), %eax  # Из первой константы в стеке кладем значение в eax
  movl %eax, %edi # Кладем значение eax в edi т.к. это нужно для вызова функции на Си
  call print # вызываем функцию print
```

movl $0, %eax # обнуляем значение возврата
leave # востанавливаем стек тут необязательно
ret # завершаем программу

Сборка и запуск программы на ассемблере выполняется так же, как и программы на Си

gcc print.c main.s
./a.out

В результате исполнения программы получен следующий вывод

Вывод
1123

Третьим способом применения ассемблерных команд в программе на языке Си является использование ассемблерных вставок. В пункте j.5.10 стандарта языка программирования Си указывается синтаксис ключевого слова asm для вставки ассемблерных команд в текст программы на языке Си.

Далее приведен пример использования ассемблерной вставки в исходном тексте программы в файле 05.c для компиляции gcc (рисунок 1).

Для компиляции файла 05.c необходимо дополнительно указать компилятору gcc нотацию, использованную для написания ассемблерной вставки

gcc -masm=intel 05.c

Компилятор gcc имеет специальный ключ, который позволяет из текста на языке Си генерировать текст на языке ассемблер. Для примера взять исходный текст

программы из пункта 6.6. Для генерации текста на языке ассемблер использована команда

gcc 01.c -S

05.c

```c
#include <stdio.h>

int main() {
    int a = 1;
    int b = 2;
    int c;

    asm(".intel_syntax noprefix\n\t"// использование синтаксиса Intel
        "mov eax, %1\n\t"
        "add eax, %2\n\t"
        "mov %0, eax\n\t"
        :"=r"(c)                    // список выходных значений
        :"r"(a), "r"(b)             // список входных аргументов
        : "eax"                     // список разрушаемых регистров
    );

    printf("%d + %d = %d\n", a, b, c);
    return 0;
}
```

Рисунок 1 — Файл 05.c

Ниже представлены первые 10 строк функции main, сгенерированные компилятором gcc в нотации AT&T.

```
main:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
```

```
movl    $2, -20(%rbp)
leaq-20(%rbp), %rax
movq    %rax, -8(%rbp)
```

Для генерации текста на языке masm, необходимо явно указать нотацию языка ассемблер с помощью директивы

```
g++ 1.cpp -S -masm=intel
```

Первые 10 строк функции main, сгенерированные в нотации intel

```
main:
.LFB1:
    .cfi_startproc
    push    rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movrbp, rsp
    .cfi_def_cfa_register 6
    sub rsp, 32
    movDWORD PTR [rbp-20], 2
    lea rax, [rbp-20]
    movQWORD PTR [rbp-8], rax
```

Технологии дизасемблирования относятся к реверсинжинирингу, т.е. к способам восстановления принципов функционирования готового продукта. Одна из популярных технологий в этой области — программа nm, позволяющая просматривать таблицы имен в бинарных объектах ОС Линукс.

Для демонстрации возможностей инструментов дезасемблирования будет использован исполняемый

файл, полученный из исходного текста программы в пункте 6.6. Программа nm и программы пакета **binutils**.

Получение имен объектов исполняемого файла a.out

nm -D a.out

w __gmon_start__
U __libc_start_main
U printf

Полный список объектов файла .a.out

nm a.out
0000000000601028 B __bss_start
0000000000601028 b completed.7557
0000000000601028 D __data_start
0000000000601028 W data_start
0000000000400460 t deregister_tm_clones
0000000000400e0 t __do_global_dtors_aux
0000000000600e18 t
__do_global_dtors_aux_fini_array_entry
0000000000400618 R __dso_handle
0000000000600e28 d _DYNAMIC
0000000000601028 D _edata
0000000000601030 B _end
0000000000400604 T _fini
0000000000400500 t frame_dummy
0000000000600e10 t __frame_dummy_init_array_entry
0000000000400a0 r __FRAME_END__
*0000000000601000 d _GLOBAL_OFFSET_TABLE_*
w __gmon_start__
0000000000400650 r __GNU_EH_FRAME_HDR
0000000000400c8 T _init
0000000000600e18 t __init_array_end
0000000000600e10 t __init_array_start

```
0000000000400610 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000600e20 d __JCR_END__
0000000000600e20 d __JCR_LIST__
                 w _Jv_RegisterClasses
0000000000400600 T __libc_csu_fini
0000000000400590 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000400542 T main
0000000000400526 T max_value
                 U printf@@GLIBC_2.2.5
00000000004004a0 t register_tm_clones
0000000000400430 T _start
0000000000601030 D __TMC_END__
```

Получение дезасемблированного текста программы на языке ассемблер возможно из исполняемого файла. Для этого можно использовать любой тип исполняемого файла как с информацией для отладки, так и без неё.

```
objdump -d a.out
```

Далее приведены первые 10 строк дезассемблированной функции main файла a.out.

```
0000000000400542 <main>:
  400542:    55                    push   %rbp
  400543:    48 89 e5              mov    %rsp,%rbp
  400546:    48 83 ec 20                sub    $0x20,%rsp
  40054a:    c7 45 ec 02 00 00 00  movl   $0x2,-
0x14(%rbp)
  400551:    48 8d 45 ec                lea    -0x14(%rbp),
%rax
```

```
400555:     48 89 45 f8              mov    %rax,-
0x8(%rbp)
400559:     c7 45 f0 03 00 00 00     movl   $0x3,-
0x10(%rbp)
400560:     48 8b 45 f8              mov    -0x8(%rbp),
%rax
400564:     8b 00                    mov    (%rax),%eax
400566:     8b 55 f0                 mov    -0x10(%rbp),%edx
```

Для просмотра содержимого динамических библиотек используется программа readelf, например, следующим образом

readelf -l libao.so.4

Будет отображена информация, приведенная на рисунке 2.

Следует иметь в виду, преобразование из Си в ассемблер возможно, но операторы Си не всегда имеют прямое отражение на ассемблерные инструкции из-за выполняемой компилятором оптимизации. В связи с этим, обратное преобразование ассемблерного текста в Си почти всегда невозможно. Преобразование между двоичным файлом и ассемблерными командами возможно в оба направления. При восстановлении программы из исполняемого файла в ассемблерный текст теряются идентификаторы переменных.

```
Тип файла ELF – DYN (Совм. исп. объектный файл)
Точка входа 0x1e80
Имеется 7 заголовков программы, начиная со смещения 64

Заголовки программы:
  Тип             Смещ.             Вирт.адр           Физ.адр
                  Рзм.фйл           Рзм.пм              Флаги  Выравн
  LOAD            0x0000000000000000 0x0000000000000000 0x0000000000000000
                  0x00000000000077ac 0x00000000000077ac  R E    200000
  LOAD            0x0000000000007be8 0x0000000000207be8 0x0000000000207be8
                  0x0000000000000900 0x0000000000000a38  RW     200000
  DYNAMIC         0x0000000000007dc8 0x0000000000207dc8 0x0000000000207dc8
                  0x00000000000001d0 0x00000000000001d0  RW     8
  NOTE            0x00000000000001c8 0x00000000000001c8 0x00000000000001c8
                  0x0000000000000024 0x0000000000000024  R      4
  GNU_EH_FRAME    0x0000000000006be8 0x0000000000006be8 0x0000000000006be8
                  0x00000000000001ec 0x00000000000001ec  R      4
  GNU_STACK       0x0000000000000000 0x0000000000000000 0x0000000000000000
                  0x0000000000000000 0x0000000000000000  RW     8
  GNU_RELRO       0x0000000000007be8 0x0000000000207be8 0x0000000000207be8
                  0x0000000000000418 0x0000000000000418  R      1

 Соответствие раздел-сегмент:
  Сегмент Разделы...
   00     .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version
_d .gnu.version_r .rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_h
dr .eh_frame
   01     .ctors .dtors .jcr .data.rel.ro .dynamic .got .got.plt .data .bss
   02     .dynamic
   03     .note.gnu.build-id
   04     .eh_frame_hdr
   05
   06     .ctors .dtors .jcr .data.rel.ro .dynamic .got
```

Рисунок 2 — Вывод утилиты readelf

6.8. Использование библиотек
6.8.1. Библиотечные функции

В программах на Си часто используется механизм, предоставляемый ОС для выделения повторяющихся функций и общих ресурсов в двоичное хранилище, отдельное от исполняемого файла и называемое библиотеками.

Библиотеки, описанные в стандарте языка программирования, называются стандартными и содержать набор функций, перечисленных в стандарте языка. В пункте 4.3.2 рассматриваются функции для работы со строками, которые составляют стандартную библиотеку, описанную в заголовочном файле string.h. Библиотека для работы со строками описана в разделе 7.24 стандарта языка программирования.

Значение библиотек в языке Си настолько велико, что их описание начинается с раздела 7 стандарта языка программирования и занимает около ¾ всего стандарта.

Кроме стандартных библиотек важный вклад вносят библиотеки, предоставляющие интерфейсы к механизмам по стандарту POSIX. Все взаимодействия с основными элементами ОС выполняются именно с помощью таких библиотек.

Для демонстрации работы с библиотеками, не входящими в стандарт языка программирования, выбраны библиотеки, описанные заголовочными файлами dirent.h и sys/stat.h, которые используются для работы с элементами файловой системы. Использован модифицированный фрагмент исходного текста программы, зарегистрированной с именем «ФС идентификация» с номером №2018663874, выданной Федеральной службой по интеллектуальной собственности.

Пояснения к исходному тексту программы даны в виде комментариев к строкам, выражения в которых используют объявления заголовочных файлов dirent.h или sys/stat.h.

01.c

```c
#include <stdio.h>
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>

#define PATH "./"

int main () {
  DIR *dir; /* Структура DIR объявлена в dirent.h */
  struct dirent *ent; /* Структура dirent объявлена в dirent.h */
  struct stat statbuf; /* Структура stat объявлена в sys/stat.h */
  dir = opendir(PATH); /* Функция opendir объявлена в dirent.h */
  while ((ent = readdir(dir)) != 0) { /* Функция readdir объявлена в dirent.h */
    if ((strcmp(ent->d_name, ".") != 0) &&
      (strcmp(ent->d_name, "..") != 0)) {
      memset((void*)&statbuf , 0, sizeof(statbuf));
      char full_path[1000] = {0};
      strcpy((char *)&full_path, PATH);
      strcat((char *)&full_path, ent->d_name);
      stat((char *)&full_path, &statbuf);
      printf("%s%s\tsize: %ld\n", PATH, ent->d_name, statbuf.st_size);
    }
  }
  closedir(dir); /* Функция closedir объявлена в dirent.h */
  return 0;
}
```

Рисунок 1 — Исходный текст программы отображения имен файлов и размера их содержимого в текущем каталоге

После создания исполняемого файла и его запуска был получен вывод

./01.c size: 1076
./a.out size: 14488

Заголовочные файлы библиотек расположены в каталоге /usr/include, а сами библиотеки в каталогах /usr/lib, /usr/lib64 или других. Следует учесть, что однозначное соответствие между именем заголовочного файла и именем библиотеки может отсутствовать.

6.8.2 Статические библиотеки

Механизмы ОС предусматривают возможность создания двух видом библиотек: статических и динамических. Они отличаются способом загрузки и выгрузки. Статические библиотеки загружаются одновременно с запуском исполняемого файла и остаются загруженными до тех пор, пока выполняется программа.

Для создания статической библиотеки необходимо использовать следующие программы:

- компилятор gcc для компиляции;
- архиватор, создающий статические библиотеки ar;
- программу создания оглавления библиотеки для ускорения вызовов функции ranlib.

Для примера создания статической библиотеки используется три файла

lib.h

```
void hello(void);
```

lib.c

```
#include <stdio.h>

void hello(void){
   printf("Hello, library world.\n");
}
```

main.c

```
#include "lib.h"

int main(){
   hello();
   return 0;
}
```

Кратчайший способ создания статической библиотеки – выполнение команд

```
gcc -c lib.c
ar cru lib.a lib.o
ranlib lib.a
```

В результате первой командой будет создан объектный файл lib.o. Вторая команда создаст файл библиотеки lib.a, а третья обновит файл lib.a, создав таблицу вызываемых функций. Для добавления статической библиотеки в исполняемый файл необходимо при компиляции указать компоновщику путь к каталогу с библиотеками и имя используемой библиотеки

```
gcc main.c -L. lib.a
```

Ключ компилятора -L. (последний символ точки) указывает на то, что библиотеки находятся в текущем каталоге. Далее указывается имя собранной ранее статической бибилотеки lib.a.

Запуск созданного исполняемого файла с именем по умолчанию производится как обычно

```
./a.out
```

Файл статической библиотеки используется только на этапе сборки программы. После этого наличие файла lib.a для запуска программы не обязательно.

Статические библиотеки являются дополнительным средством структурирования программы в процессе разработки. Их применение не требует дополнения исходного текста программы.

6.8.3 Динамические библиотеки

Динамическая библиотека запускается в момент её вызова программой. Для её загрузки необходимо вызвать специальную функцию загрузки библиотеки и функцию получения адреса требуемой функции. ОС ведет счетчик используемых процессов. Динамическая библиотека выгружается, если ни один процесс её не использует. Файл библиотеки требуется поставлять совместно с исполняемым файлом.

Создание динамической библиотеки требует немного больше усилий по сравнению со статической. Для создания динамической библиотеки будут использованы файлы

lib.h

```
void hello(void);
```

lib.c

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

void hello(void){
   printf("Hello, library world.\n");
}
```

Для компиляции динамической библиотеки сначало необходимо создать объектный файл командой

```
gcc -fPIC -c lib.c
```

Ключ -fPIC позволяет создать объектный файл,

содержащий код, независимый от расположения в файле. Такой эффект достигается использованием относительной адресации функций. Для получения более подробной информации по этому вопросу необходимо обратиться к описанию PIC (Position Independent Code) технологии исполняемых файлов.

Сборка динамической библиотеки производится с помощью команды

gcc -shared -o lib.so.0.0 lib.o -lc

Ключ -shared сообщает компилятору о необходимости создания динамической библиотеки. Ключ -lc указывает на необходимость использования стандартной библиотеки libc, включающей в себя реализацию функции stdio.h. Ключ -o задает имя выходного файла.

Для интеграции созданной динамической библиотеки в ОС необходимо **скопировать** её в каталог /lib или lib64 в зависимости от архитектуры ОС.

Следующая команда выполняет настройку связывания времени исполнения в динамическом компоновщике, т.е. объясняет ОС, где искать каталоги с динамическими библиотеками. Комманда выполняется только **после** копирования библиотеки в каталог /lib или /lib64

/sbin/ldconfig -v -n

Такую привязку допустимо выполнять редактированием файла /etc/ld.so.conf . В современных дистрибутивах этот файл содержит ссылку на каталог с файлами, содержащими пути к библиотекам. Например: include /etc/ld.so.conf.d/*.conf , где и нужно размещать

файлы, содержащие пути к библиотекам. В файле /etc/ld.so.cache хранится кеш путей и библиотек в двоичном виде. Для более подробного изучения ldconfig необходимо воспользоваться встроенной справкой man. Для этого следует выполнить команду man ldconfig.

main.c

```c
#include <stdio.h>
#include <dlfcn.h> /* dlopen, dlsum, dlclose */
#include <stdlib.h>
#include "lib.h"

typedef void (*function)(void);

int main(void) {
    const char *error;
    void *dl;
    function dl_function;
    /* Загрузка библиотеки */
    dl = dlopen("lib.so.0.0", RTLD_LAZY);
    if (!dl) {
        fprintf(stderr, "lib.so.0.0 error: %s\n", error);
        exit(1);
    }
    /* Получение указателя на функцию. */
    dlerror();
    dl_function = dlsym(dl, "hello");
    if ((error = dlerror())) {
        fprintf(stderr, "Couldn't find hello: %s\n", error);
        exit(2);
    }
    (*dl_function)(); /* Вызов функции из библиотеки */
    dlclose(dl); /* Закрытие библиотеки, уменьшения счетчика обращения к ней. */
    return 0;
}
```

Рисунок 1 – Исходный текст программы, вызывающей динамическую библиотеку

Для просмотра кеша динамических библиотек используется команда

/sbin/ldconfig -p

Для этого необходимо добавить в файл пути, в которых следует искать динамические библиотеки.

Исходный текст программы, вызывающей функцию из динамически подключаемой библиотеки, представлен на рисунке 1.

Получение объектного файла из исходного текста программы выполняется как обычно

gcc -c main.c

При компановке исполняемого файла необходимо указать дополнительный ключ

gcc main.o -ldl

Дополнительный ключ необходим для включения в исполняемый файл библиотеки, содержащей функции для работы с динамическими библиотеками (dlopen, dlsym, dlclose).

Провести сборку исполняемого файла возможно одной командой

gcc main.c -ldl

Если библиотека не была скопирована в системную папку для динамических библиотек такого типа и находится в том же каталоге, что и файл программы, запуск полученного исполняемого файла производится следующим образом

LD_LIBRARY_PATH="." ./a.out

или

export LD_LIBRARY_PATH="."
./a.out

После запуска программы должен быть получен следующий вывод

Hello, library world.

При поставке программного обеспечения в качестве двоичных файлов, в случае отсутствия одной из библиотек требуется установить, какая именно библиотека была недопоставлена или удалена. Для этого подходит команда

ldd a.out

Вывод команды

linux-vdso.so.1 (0x00007fff69baf000)
libc.so.6 => /lib64/libc.so.6 (0x00007fc41aba7000)
/lib64/ld-linux-x86-64.so.2 (0x00007fc41b14c000)

Информация об используемых библиотеках запущенных процессов храниться в файле maps ФС /proc/ и может быть отображена командой

cat /proc/$PID/maps

Соответственно, для использования этой библиотеки необходимо наличие еще трех библиотек.

При запуске ОС динамической библиотеки производится выполнение функции _init(). При выгрузке библиотеки выполняется функция _fini(). Следует

обратить внимание, функции вызываются только при первом запуске программы. Если библиотеку вызывают другие программы, повторного вызова функции инициализации не произойдет. Если первая программа успеет завершить работу с библиотекой до того, как библиотеку запросит вторая программа, после запроса второй программы библиотека будет загружена заново и функция _init() выполнится еще раз. По такому же принципу работает функция _fini().

Для демонстрации работы функций _init(), _fini() необходимо изменить файлы lib.h и lib.c следующим образом

lib.h

```
void _init(void);
void hello(void);
void _fini(void);
```

lib.c

```
#include <stdio.h>

void _init(void){
   printf("Load library!\n");
}

void hello(void){
   printf("Hello, library world.\n");
}

void _fini(void){
   printf("Library destroy!\n");
}
```

Компиляция файлов производится таким же образом

gcc -fPIC -c lib.c

Компиляция должна пройти без ошибок. При сборке библиотеки командой будут получены следующие сообщения об ошибках

lib.o: In function `_init':
lib.c:(.text+0x0): multiple definition of `_init'
/usr/lib64/gcc/x86_64-alt-linux/5/../../../../lib64/crti.o:/usr/src/RPM/BUILD/glibc-2.23-alt3.M80P.2/csu/../sysdeps/x86_64/crti.S:64: first defined here
lib.o: In function `_fini':
lib.c:(.text+0x26): multiple definition of `_fini'
/usr/lib64/gcc/x86_64-alt-linux/5/../../../../lib64/crti.o:/usr/src/RPM/BUILD/glibc-2.23-alt3.M80P.2/csu/../sysdeps/x86_64/crti.S:64: first defined here
collect2: ошибка: выполнение ld завершилось с кодом возврата 1

Ошибка сообщает, что функции _init() и _fini() определены в файле crti.S . Для определения библиотеки, в которой находится реализация данной функции, сборку необходимо выполнить с дополнительным ключом -v

gcc -shared -o lib.so.0.0 lib.o -lc -v

В выводе этой команды в конце присутствуют строки

lib.c:(.text+0x0): multiple definition of `_init'
/usr/lib64/gcc/x86_64-alt-linux/5/../../../../lib64/crti.o:/usr/src/RPM/BUILD/glibc-2.23-alt3.M80P.2/csu/../sysdeps/x86_64/crti.S:64: first defined here
lib.o: In function `_fini':
lib.c:(.text+0x26): multiple definition of `_fini'
/usr/lib64/gcc/x86_64-alt-linux/5/../../../../lib64/crti.o:/usr/src/RPM/BUILD/glibc-2.23-alt3.M80P.2/csu/../sysdeps/x86_64/crti.S:64: first defined here

Вывод сообщает нам о том, что фунции реализованы в библиотеке crti.o. Для создания библиотеки необходимо выполнить программу collect2 с ключами, сгенерированными компилятором gcc, удалив предварительно путь к библиотеке crti.o. Была получена следующая строка

/usr/libexec/gcc/x86_64-alt-linux/5/collect2 -plugin /usr/libexec/gcc/x86_64-alt-linux/5/liblto_plugin.so -plugin-opt=/usr/libexec/gcc/x86_64-alt-linux/5/lto-wrapper -plugin-opt=-fresolution=/tmp/.private/valerii/ccnidWIZ.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --no-add-needed --eh-frame-hdr --hash-style=gnu --as-needed -m elf_x86_64 -shared -o lib.so.0.0 /usr/lib64/gcc/x86_64-alt-linux/5/crtbeginS.o -L/usr/lib64/gcc/x86_64-alt-linux/5 -L/usr/lib64/gcc/x86_64-alt-linux/5/../../../../lib64 -L/lib/../lib64 -L/usr/lib/../lib64 -L/usr/lib64/gcc/x86_64-alt-linux/5/../../.. lib.o -lc -lgcc --as-needed -lgcc_s -lc -lgcc --as-needed -lgcc_s /usr/lib64/gcc/x86_64-alt-linux/5/crtendS.o /usr/lib64/gcc/x86_64-alt-linux/5/../../../../lib64/crtn.o

В файл main.c изменений вносить не требуется. Его сборка осуществляется как обычно

gcc main.c -ldl

После сборки программы её запуск производится как обычно

LD_LIBRARY_PATH="." ./a.out

В результате был получен вывод

Load library!
Hello, library world.
Library destroy!

Таким образом была произведена замена функций _init() и _fini() в библиотеке lib.

Также для отключения сборки стартовых функций, в которых присутствуют _init() и _fini(), по умолчанию можно использовать ключ компилятора gcc -nostartfiles.

7. Экспресс — алгоритмизация
7.1. Блок-схемы и UML

Во введении приводится краткое описание нескольких типов мышления, которые необходимо развивать программисту для выполнения своих профессиональных обязанностей. Все они базируются на алгоритмическом типе мышления, который является главным для парадигмы языка программирования Си. Его формирование занимает не менее шести месяцев. Такие типы мышления как объектный, параллельный (связанный с многопоточным программированием) можно развить за месяц, но только при условии наличия алгоритмического типа мышления. Базовые понятия для понимания алгоритмов приведены в пункте 1.3.

Алгоритмы могут быть записаны в любом формализованном виде. Под этим подразумевается, в том числе, текстовая запись в виде комментариев в исходном тексте программы, которые не требуют уточнения при написании исходного текста программы. В классическом понимании, формализованный вид — это описание с помощью математических формул.

Наиболее популярный вид представления формализованных алгоритмов — блок-схема. Блоки, т.е. составные части и правила составления блок-схем описаны в ГОСТ 19.701-90 «Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения». Развитие алгоритмического мышления путем построения блок-схем выходит за рамки этой книги. Дополнительную информацию о правилах построения блок-схем можно найти в Википедии. Классическим произведением в этой области является трехтомник Кнут Д. Э. «Искусство программирования».

Умение читать и составлять блок-схемы является

обязательным навыком для программиста использующего язык Си. Многие аппаратные и программные стандарты содержат пояснения к своей работе, выполненные в виде блок-схем.

Для программирования с использованием других типов мышления используется UML. Он применяется для создания абстрактных моделей и позволяет описывать классы, компоненты, архитектуры, а также такие процессы как обобщение, агрегацию.

7.2. Реализация односвязных и многосвязных списков

Для обсуждения эффективности и производительности алгоритмов в пункте 8.3 предлагается использовать программу, оперирующую связным списком. В структурах данных этого типа для каждого элемента выделяются ячейки памяти в ОЗУ. Связные списки бывают разного вида. Они отличаются количеством указателей на другие элементы в каждом элементе, а также необходимостью создавать отдельные указатели на начало и конец связного списка.

В языке Си есть набор «лучших практик», которые рекомендованы к использованию. В отношении работы со списками такой лучшей практикой является применение заголовочного файла /usr/include/sys/queue.h . Файл содержит набор макросов, выполняющие действия с указателями, т. е. операции, которые являются основным источником ошибок при использовании связных списков. Файл написан в 1991-1994 годах в университете Беркли.

В таблице 1 указано, какие связные списки реализованы в файле queue.h .

Таблица 1 — Соответствие функций для работы со списками типам связных списков

Типы соответствий:

+ макрос для данного типа списка доступен;

- макрос для данного типа списка недоступен;

s макрос доступен, выполняется медленно (время исполнения O(n)).

МАКРОС \ ФУНКЦИЯ SLIST LIST STAILQ TAILQ

Таблица 1. продолжение

_HEAD	+	+	+	+	
_CLASS_HEAD		+	+	+	+
_HEAD_INITIALIZER		+	+	+	+
_ENTRY	+	+	+	+	
_CLASS_ENTRY		+	+	+	+
_INIT		+	+	+	+
_EMPTY	+	+	+	+	
_FIRST	+	+	+	+	
_NEXT	+	+	+	+	
_PREV	-	+	-	+	
_LAST	-	-	+	+	
_FOREACH		+	+	+	+
_FOREACH_FROM		+	+	+	+
_FOREACH_SAFE		+	+	+	+
_FOREACH_FROM_SAFE		+	+	+	+
_FOREACH_REVERSE			-	-	-
+					
_FOREACH_REVERSE_FROM		-	-	-	+
_FOREACH_REVERSE_SAFE		-	-	-	+
_FOREACH_REVERSE_FROM_SAFE			-	-	-
+					
_INSERT_HEAD		+	+	+	+

Продолжение таблицы 1

_INSERT_BEFORE		-	+	-	+
_INSERT_AFTER		+	+	+	+
_INSERT_TAIL		-	-	+	+
_CONCAT		s	s	+	+
_REMOVE_AFTER		+	-	+	-
_REMOVE_HEAD		+	-	+	-
_REMOVE		s	+	s	+
_SWAP	+	+	+	+	

Для реализации программы используется однонаправленный связный список типа SLIST. Однонаправленный список имеет функции перемещения от начала списка к его концу, однако не позволяет возвращаться на предыдущий элемент. Кроме того, функции _REMOVE выполняются медленнее, чем в списке LIST. Без функции _REMOVE список пригоден для наполнения неизменными данными для использования в качестве справочника, но не пригоден для выполнения интенсивных операций с его элементами.

main.c

```c
#include <sys/queue.h>
#include "stdlib.h"
#include "stdio.h"

typedef struct ListItem {
    SLIST_ENTRY(ListItem) next;
    int data;
} ListItem;

typedef SLIST_HEAD(ListHead, ListItem) ListHead;

int main(int argc, char *argv[]) {
    int data = 0;
    ListItem * ni;
    ListHead * list_head;
    list_head = malloc(sizeof(ListHead));
    SLIST_INIT(list_head);
    /*Добавление в head*/
    int i;
    for (i = 0; i < 10; i++) {
        ni = malloc(sizeof(ListItem));
        ni->data = i;
        SLIST_INSERT_HEAD(list_head, ni, next);
    }
    /*Вывод на экран*/
    fprintf(stdout, "Data input: ");
    SLIST_FOREACH(ni, list_head, next) {
        fprintf(stdout, "%d ", ni->data);
    }
```

Рисунок 1 — Первая часть исходного текста программы

На рисунке 1 представлены первые 29 строк исходного текста программы, выполняющей работу со списками.

На рисунке 2 представлено окончание исходного текста программы.

```
30      fprintf(stdout, "\n");
31      /*Удалить значение 5*/
32      data = 5;
33      SLIST_FOREACH(ni, list_head, next) {
34          if (ni->data == data)
35              SLIST_REMOVE(list_head, ni, ListItem, next);
36      }
37      /*Вывод на экран*/
38      fprintf(stdout, "5 deleted: ");
39      SLIST_FOREACH(ni, list_head, next) {
40          fprintf(stdout, "%d ", ni->data);
41      }
42      fprintf(stdout, "\n");
43      /*Освобождаем память*/
44      while (list_head->slh_first != NULL) {
45          ni = list_head->slh_first;
46          SLIST_REMOVE_HEAD(list_head, next);
47          free((void*)ni);
48      }
49      /*Вывод на экран*/
50      fprintf(stdout, "Items destroy: ");
51      SLIST_FOREACH(ni, list_head, next) {
52          fprintf(stdout, "%d ", ni->data);
53      }
54      fprintf(stdout, "\n");
55      free(list_head);
56      fprintf(stdout, "Exit\n");
57      return EXIT_SUCCESS;
58  }
59
```

Рисунок 2 — Окончание исходного текста программы

Для реализации однонаправленного связного списка в программе используется два типа:

- ListHead — структура с данными, необходимыми

для работы всего списка;

- ListItem — для элемента списка.

Тип ListHead является переопределением типа SLIST_HEAD(ListHead, ListItem), который необходим для работы макросов, обеспечивающих действия со списками. Переопределение выполнено в строке 10. В программе используется один список, поэтому переменная этого типа также одна и имеет идентификатор list_head.

Тип ListItem содержит в себе указатель на следующий элемент и данные типа int с идентификатором data, которые и являются «полезной нагрузкой» элемента списка. Объявление списка произведено в строках 5-8. В алгоритмах программы используется переменная ni.

Программа собирается из этого исходного текста командой

gcc main.c

В результате выполнения программы получен следующий вывод:

```
Data input: 9 8 7 6 5 4 3 2 1 0
5 deleted: 9 8 7 6 4 3 2 1 0
Items destroy:
Exit
```

Первая строка вывода сформирована строками 25-30. До этих строк выполняется подготовительная работа по созданию необходимых элементов и структур. В строках 31-36 производится удаление элемента со значением поля data = 5. В строках 37-42 производится повторный вывод списка с удаленным элементом со значением 5. В

строках 43-57 выполняется освобождение ресурсов, вывод соответствующих сообщений и подготовка к завершению программы.

В стандарте языка программирования в пункте 6.7.2.3 приводится пример описания структуры для связных списков, однако данная тема является объемной и не будет рассматриваться в этой книге. Она включает в себя не только набор приемов по работе со связными списками, но и изучение алгоритмов по эффективной обработке данных, хранимых в связных списках.

7.3 Приемы оценки вычислительной сложности алгоритмов

Эффективность алгоритмов является одной из сложных тем программирования. В учебных заведениях редко имеются объемные курсы по оптимизации алгоритмов, т.к. овладение эффективными приемами оптимизации требует значительного времени и обычно приходит с многолетним опытом разработки.

С другой стороны оптимизация в процессе разработки любых технических решений является обычным делом. Оптимизация расхода некоторого ресурса на 12% считается нормой для любой области технологий. Исключение составляют высокооптимизированные решения с многолетней историей. К ним относятся многие узлы техники, выпускаемой серийно.

Программирование предоставляет еще больше возможностей для повышения производительности. Личным рекордом автора этой книги стало повышение скорости алгоритма в 595 раз или на 59500%. Это стало возможно с помощью помещения всех данных в ОЗУ и отказом от использования операций по выделению и удалению участков памяти ОЗУ в процессе вычислений, которые использовались весьма интенсивно.

Производители современных микропроцессоров выполняют расширение набора аппаратных инструкций быстрее, чем производители компиляторов успевают адаптировать под них алгоритмы оптимизации. У бюджетной модели микропроцессора Intel(R) Celeron(R) 2955U @ 1.40GHz значение поля flags, выводимое при вызове cat /proc/cpuinfo, содержит названия десятков аппаратных расширений микропроцессора, большинство из которых популярные программы не используют.

Все вышесказанное демонстрирует важность как аппаратного, так и программного обеспечения для эффективного решения вычислительных задач. Под эффективностью в данном случае понимается близкое к оптимальному потребление наиболее ограниченного ресурса, например возможности сетевого интерфейса или объем используемого ОЗУ.

Оценка потребления ресурсов является отдельной задачей. При оценке эффективности алгоритма обычно достаточно определить, потребляется ли ресурс полностью и не возникает ли исключительных ситуаций из-за превышения в его потребности. В ОС реального масштаба времени перегрузка любого ресурса приведет к останову ОС с возможной перегрузкой. В ОС, используемых в качестве домашних и офисных ЭВМ, такие режимы не используются.

Возвращаясь к оценке эффективности алгоритмов, необходимо обозначит **самый главный критерий эффективности** алгоритма – возможность получить правильный результат. Следующие по важности и представляющие наибольшую трудоемкость составляет оценка эффективности использования **памяти ОЗУ** и **времени исполнения** алгоритма и их оптимизация. В общем случае после разработки программы замеряется продолжительность исполнения отдельных функций в программе, в которых потребление времени или ОЗУ не сопоставимо велико по сравнению с остальными функциями ПО. Для каждого из таких мест принимается решение о возможности оптимизации.

При выполнении оптимизации времени исполнения алгоритма общепринятым подходом является оценка минимального, среднего и максимального времени его исполнения. Для этого проводится сравнение продолжительности работы как самого алгоритма с

разными наборами данных, так и различных алгоритмов в прочих равных условиях. К условиям выполнения алгоритма здесь относится производительность ЭВМ, инструментарий, используемый для создания программы из исходных текстов, язык программирования. Также считается, что все данные, необходимые для выполнения алгоритма, загружены в ОЗУ.

Определение вычислительной эффективности алгоритма заключается в поиске ответов на следующие вопросы:

- как зависит время работы программы от роста объема исходных данных;

- как быстро растет время работы в зависимости от роста объема входных данных, т.е. своего рода первая производная от первого вопроса.

Оценка производительности алгоритмов производится для трех случаев:

1) при минимальном количестве исходных данных;

2) при максимальном количестве данных;

3) при любом другом количестве данных.

Эти оценки называются оценочными границами или симптотическими обозначениями. Они вычисляются не точно, т.к. на протекание вычислительного процесса влияет множество факторов. Время исполнения алгоритма вычисляется по аргументу с самой большой степенью.

Зависимость продолжительности вычислений от количества входных данных математически обозначается как Θ(n) (тетта от n). Под данным обозначением подразумеваются все возможные случаи.

Для обозначения наилучшей и наихудшей производительности используют следующие обозначения:

- Ω (n) (амега от n) дает нижную границу, т.е. Θ(1) =

Ω(1) наилучший результат, который нельзя превзойти без смены алгоритма;

- O(n) (омикрон от n), при n = max, дает наихудший результат, т.е. Θ(n) = O(n), n = max .

Наиболее распространенными формами Θ(n) являются:

- квадратичная $y=\Theta(n^2)$,
- логарифмическая $y=\Theta(n*lg(n))$, где lg — это алгоритм по основанию 2, т.е. $lg(n)=\log_2(n)$,
- линейная $y=\Theta(n)$.

Оценку производительности допустимо выполнять не только для программы в целом или функции, но и для произвольной части исходного текста программы. Для примера произведем оценку производительности фрагмента программы, получаемой с помощью строк вывода элементов связного списка из пункта 7.2. Этот фрагмент исходного текста расположен в файле main.c в строках 25-29, и повторяется в строках 37-41, 49-53.

Фрагмент исходного текста

```
/*Вывод на экран*/
fprintf(stdout, "Data input: ");
SLIST_FOREACH(ni, list_head, next) {
   fprintf(stdout, "%d ", ni->data);
}
fprintf(stdout, "\n");
```

Пусть оператор printf() выполняется за время t независимо от аргументов и макрос SLIST_FOREACH также выполняется за t. В этом случае время выполнения этого фрагмента текста равно

$t + (t + t)*n + t = \Theta(n)$, где n — количество итераций цикла.

$$\Theta(n) = 2t + 2tn = 2t\,(n+1)$$

Таким образом можно произвести расчет Θ(n) для некоторых значений n, а также оценочных границ этого алгоритма. Нижняя граница в случае отсутствия элементов в связном списке Ω (0) = 2t (0+1) = 2t+1. Делаем допущение о том, что максимальное значение n = 10. В этом случае верхняя граница O(n) = 2t (10+1) = 22t.

Необходимо ответственно подходить к изучению алгоритмизации и исследованию эффективности алгоритмов. Приведем пример частого заблуждения среди неопытных программистов. Во многих источниках алгоритм сортировки qsort описывается как наиболее эффективный, имеющий время исполнения от $\Theta(n^2)$ до $\Theta(n*lg(n))$. Однако, чем меньше данных нужно отсортировать, тем медленнее он будет работать. В случае сортировки одного значения, классический пузырьковый метод дает лучшее время, чем qsort.

Другим интересным примером является сортировка подсчетом. Её смысл заключается в использовании вспомогательного массива всех возможных значений. Во время сортировки ведется подсчет количества элементов для каждого возможного значения. В конце каждый элемент выводится столько раз, сколько раз он встретился при подсчете. Данный алгоритм работает быстрее, чем обычная сортировка сравнением, при которой каждый элемент сравнивается с другими элементами массива. Сортировка подсчетом применима только для небольшого набора возможных значений.

Наиболее полным учебником по алгоритмизации является классическое произведение — 4-х томник Дональда Кнута Искусство программирования.

8. Новый взгляд на программирование, «Привет, мир!»

Современное программирование — это не только написание текста программы и преобразование его в исполняемый файл для ОС. Причиной этого стало повышение сложности разрабатываемых программ и увеличение количества строк в них. Сегодня редкое коммерческое приложение имеет меньше 10 000, или даже 100 000 строк исходного текста. Разработка таких проектов значительно упрощается при использовании средств автоматизации на всех этапах создания программы. Рассмотрим учебный пример использования основных технологий разработки программ.

Продемонстрировать сложный проект не представляется возможным, поэтому для учебного примера будет использована программа с минимальным объемом исходного текста программы.

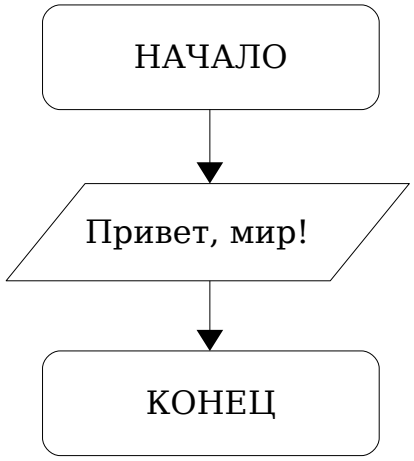
Этап 1. Постановка задачи.

Самой простой задачей, с которой начинают знакомство с языком программирования, является вывод строки «Привет, мир!». Сформулируем задачу таким же образом. Условие задачи: «Вывести строку «Привет, мир!»».

Результат этого этапа — подробное описание задания, т. к. это то, что в реальных проектах называют техническим заданием или ТЗ. Также это может быть документ, заменяющий ТЗ :J .

Этап 2. Формализация.

Одной из целей формализации может быть блок-схема. Формализуем поставленную задачу.



Результатом данного этапа является блок-схема.

Этап 3. Проектирование файловой структуры проекта.

Проектирование файловой структуры проекта обычно не выполняется при использовании интегрированных сред разработки, таких как Eclipse, Visual C++. Среди использующих интегрированные среды навык проектирования ФС прививается после нескольких лет работы. Проектирование ФС обычно выполняется с использованием стандарта POSIX.

Для текущего проекта потребуется каталог проекта. Имя каталога выбрано произвольно project. В каталоге создан единственный файл main.c , который будет содержать исходный текст программы.

На этом же этапе инициализируется git с помощью команды

git init

Также рекомендуется добавить ветку dev для проведения разработки. Ветка master редко используется в сложных проектах.

git checkout -b dev

Результат этапа — файловая структура проекта.

Этап 4. Написание текста комментариев.

Сначала в файле main.c пишется условие задачи, полученное на этапе1.

```
/*!
 * \file
  * \brief Файл содержит программу вывода фразы "Привет, мир!"
 *
  * Данный файл содержит программу вывода фразы "Привет, мир!"
*/
```

Далее исходя из условия задачи описываются входные и выходные данные функций. Писать комментарий необходимо в стиле doxygen для последующей автоматической генерации документации.

```
/**
 * \brief Программа вывода фразы "Привет, мир!"
 *
 * Программа выводит фразу "Привет, мир!"
 * и завершает свое выполнение. Программа не принимает
 * аргументов.
 *
 * \param argc Количество аргументов
 * \param argv Список аргументов
 *
 * \return Статус завершения программы
 */
```

Результатом этого этапа является файл с описанием файла и функции в формате doxygen.

Этап 5. Подготовка к компиляции и сборке.

На этом этапе подготавливается файл Makefile. Он позволяет компилировать целый проект в многопоточном режиме одной командой.

```
hw: main.o
	gcc main.o -o hw
main.o: main.c
	gcc -pedantic -Wall --std=c17 -c main.c
```

Autotools для одного файла использован не будет.

Результатом этого этапа является Makefile, позволяющий командой make получать исполняемый файл.

Этап 6. Набор исходного текста программы.

На этом этапе производится написание программы.

```
#include <stdio.h>

int main () {
	puts("Привет, мир!");
	return 0;
}
```

Результатом этого этапа является набранный исходный текст программы.

Этап 7. Статическая проверка исходного текста программы.

Для статической проверки исходного текста программы используется cppcheck. Его применение заключается в вводе команды

cppcheck --std=c11 .

Результатом этого этапа является исходный текст программы и исправленные ошибки, обнаруженные статическим анализатором.

Этап 8. Компиляция и компоновка программы.

Данный этап позволяет проверить возможность сборки программы из исполняемого файла. Для этого выполняется команда

make

Результат этого этапа — исполняемый файл, создание которого подтверждает отсутствие ошибок на этапе компиляции и сборки.

Этап 9. Запуск для отладки.

Данный этап позволяет выполнить отладку созданной программы. Для этого вызывается отладчик и в качестве аргумента ему передается имя программы

gdb hw

Для запуска отладки нужно ввести

run

После ознакомления с отладочным выводом ввести

q

В случае нахождения ошибок, необходимо возвращаться на этап 6 и исправлять найденные ошибки.

Результатом данного этапа является программа,

успешно выполняющая все отладочные сценарии.

Этап 10. Проверка на утечки памяти.

Для поиска утечек памяти в ОЗУ необходимо выполнить команду

valgrind —leak-check=full ./hw

В случае нахождения ошибок, необходимо возвращаться на этап 6 и исправлять найденные ошибки.

Результатом этого этапа является программа, в которой все выделяемые участки памяти корректно освобождаются.

Этап 11. Выполнение профилирования программы.

Для выполнения профилирования используется команда

gcc -pg main.c -O3 -Wall -pedantic -std=c17 -o hw

Это сделано для того, чтобы не вносить исправление в Makefile. Далее программа запускается на исполнение.

./hw

В результате исполнения создается двоичный файл gmon.out. Для его преобразования в человекочитаемую форму необходимо выполнить команду

prof ./hw > 1.txt

В случае обнаружения функций, неудовлетворяющих критерию производительности, необходимо возвращаться на этап от 1 до 6, по ситуации, и исправлять найденные ошибки.

В результате выполнения данного этапа полученная программа будет обладать необходимой производительностью, т. е. не будет «тормозить».

Этап 12. Создание документации к программе.

Для создания документации необходимо выполнить две команды.

1. Создать проект doxygen.

doxygen -g

2. Создать документацию

doxygen

Результатом этого этапа станет автоматическое создание документации к программе.

Вывод: На сегодняшний день существуют технологии повышения качества программ, игнорирование которых усложняет разработку и понижает качество программы.

В приведенном примере не рассмотрены средства совместной разработки и средства управления разработкой.

Заключение

Большинство вопросов в книге рассмотрены обзорно. Затронутые в книге темы требуют годы для полноценного изучения, но и начинать работать не имея представления обо всех этих инструментах и приемах, не эффективно.

Особое внимание следует обратить на формирование алгоритмического мышления, практикуясь в алгоритмизации. Наиболее эффективно заниматься алгоритмизацией, изучая готовые решения и алгоритмические приемы с помощью четырехтомника «Искусство программирования» Дональда Кнута. Однако данное произведение имеет высокий уровень сложности. Если требуется начать с чего-то совсем простого, можно воспользоваться разделом на сайте abashin.ru в котором имеется около 100 готовых блок-схем и условий задач к ним. Для обучения алгоритмизации необходимо читать условие задачи и пытаться построить блок-схему её решения, а если не получится, можно подсмотреть готовое решение. Повторять решение задачи следует до тех пор, пока блок-схема не будет построена правильно.

Литература

1. Винер Н. Кибернетика, или управление и связь в животном и машине. 1948-1961 [Текст]. - 2-е издание. - М.: Наука; Главная редакция изданий для зарубежных стран, 1983. - 344 с.
2. Кнут Д.Э. Искусство программирования. Том 1. Основные алгоритмы = The Art of Computer Programming. Volume 1. Fundamental Algorithms [Текст] / под ред. С.Г. Тригуб (гл. 1), Ю.Г. Гордиенко (гл. 2) и И.В. Красикова (разд.2.5 и 2.6). - Москва: Вильямс, 2002. - Т.1. - 720 с.
3. Кнут Д.Э. Искусство программирования. Том 2. Получисленные алгоритмы = The Art of Computer Programming. Volume 2. Seminumerical Algorithms [Текст] / под ред. Л.Ф.Козаченко (гл.3, разд.4.6.4 и 4.7), В.Т.Тертышного (гл. 4) и И.В.Красикова (разд.4.6). - Москва: Вильямс, 2001. - Т.2. - 832 с.
4. Кнут Д.Э. Искусство программирования. Том 3. Сортировка и поиск = The Art of Computer Programming. Volume 3. Sorting and Searching [Текст] / под ред. В.Т.Тертышного (гл.5) и И.В.Красикова (гл.6). - 2-е изд. - Москва: Вильямс, 2007. - Т.3. - 832 с.
5. Кнут Д.Э. Искусство программирования, том 4, А. Комбинаторные алгоритмы, часть 1 = The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1 [Текст] / под ред. Ю.В.Козаченко. - 1. - Москва: Вильямс, 2013. - Т.4. - 960 с.
6. Макарова Н.В., Волков В.Б. Информатика: Учебник для вузов [Текст] - СПб.: Питер, 2011. - 576 с.: ил.
7. Эбен Майкл, Таймэн Брайан. FreeBSD. Энциклопедия пользователя [Текст], - 3-е изд. перераб. и доп.: Пер. с англ./Майкл Эбен, Брайан Таймэн. — СПб.: ООО

«ДиаСофтЮП», 2003. - 768 с.
8. RFC 3629 UTF-8, a transformation format of ISO 10646 [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc3629 .
9. RFC 1321 The MD5 Message-Digest Algorithm [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc1321 .
10. RFC 1489 Registration of a Cyrillic Character Set (koi8-r) [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc1489 .
11. RFC 4251 The Secure Shell (SSH) Protocol Architecture [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc4251 .
12. RFC 4716 The Secure Shell (SSH) Public Key File Format [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc4716 .
13. RFC 1951 DEFLATE Compressed Data Format Specification version 1.3 [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc1951 .
14. RFC6986 GOST R 34.11-2012: Hash Function (ГОСТ Р 34.11-2012) [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc6986 .
15. RFC 791 INTERNET PROTOCOL (IPv4) [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc791 .
16. RFC 2460 Internet Protocol, Version 6 (IPv6) [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc2460 .
17. RFC 1700 ASSIGNED NUMBERS [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc1700 .
18. RFC 1918 Address Allocation for Private Internets [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc1918 .
19. RFC 792 INTERNET CONTROL MESSAGE

PROTOCOL [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc792 .
20. RFC 950 Internet Standard Subnetting Procedure [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc950 .
21. RFC 793 TRANSMISSION CONTROL PROTOCOL [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc793 .
22. RFC 768 User Datagram Protocol [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc768 .
23. RFC 1945 Hypertext Transfer Protocol — HTTP/1.0 [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc1945 .
24. RFC 2616 Hypertext Transfer Protocol — HTTP/1.1 [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc2616 .
25. RFC 7231 Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc7231 .
26. RFC 3912 WHOIS Protocol Specification [Электронный ресурс]. - URL: https://tools.ietf.org/html/rfc3912 .
27. ISO/IEC 12207:2008 System and software engineering — Software life cycle processes. (ИСО/МЭК 12207 — 2008 Системная и программная инженерия. Процессы жизненного цикла программных средств) [Электронный ресурс]. - URL: https://www.iso.org/standard/43447.html .
28. ISO/IEC 9899:2018 Information technology -- Programming languages — C [Электронный ресурс]. - URL: https://www.iso.org/ru/standard/74528.html .
29. ISO 9660:1988 Information processing -- Volume and file structure of CD-ROM for information interchange [Электронный ресурс]. - URL:

https://www.iso.org/standard/17505.html .
30. ISO/IEC 10646:2017 Information technology -- Universal Coded Character Set (UCS) [Электронный ресурс]. - URL: https://www.iso.org/standard/69119.html .
31. ISO/IEC/IEEE 9945:2009 Information technology -- Portable Operating System Interface (POSIX®) Base Specifications, Issue 7 [Электронный ресурс]. - URL: https://www.iso.org/standard/50516.html .
32. ISO/IEC 7816-15:2016 Identification cards -- Integrated circuit cards -- Part 15: Cryptographic information application (ГОСТ Р ИСО/МЭК 7816) [Электронный ресурс]. - URL: https://www.iso.org/ru/standard/65250.html.
33. ГОСТ 28147-89 Криптография ГОСТ основные понятия [Электронный ресурс]. - URL: http://protect.gost.ru/v.aspx?control=8&baseC=-1&page=0&month=-1&year=-1&search=&RegNum=1&DocOnPageCount=15&id=131282 .
34. ГОСТ Р ИСО/МЭК 7498-1-99 ВОС. Базовая эталонная модель. Часть 1. Базовая модель [Электронный ресурс]. - URL: http://protect.gost.ru/v.aspx?control=8&baseC=-1&page=0&month=-1&year=-1&search=&RegNum=1&DocOnPageCount=15&id=124460 .
35. ГОСТ Р ИСО 7498-2-99 ВОС. Базовая эталонная модель. Часть 2. Архитектура защиты информации [Электронный ресурс]. - URL: http://protect.gost.ru/v.aspx?control=8&baseC=-1&page=0&month=-1&year=-1&search=&RegNum=1&DocOnPageCount=15&id=123561 .

36. ГОСТ Р ИСО 7498-3-97 ВОС. Базовая эталонная модель. Часть 3. Присвоение имён и адресация [Электронный ресурс]. - URL: http://protect.gost.ru/v.aspx?control=8&baseC=-1&page=0&month=-1&year=-1&search=&RegNum=1&DocOnPageCount=15&id=125245 .
37. ГОСТ Р ИСО/МЭК 7498-4-99 ВОС. Базовая эталонная модель. Часть 4. Основы административного управления [Электронный ресурс]. - URL: http://protect.gost.ru/v.aspx?control=8&baseC=-1&page=0&month=-1&year=-1&search=&RegNum=1&DocOnPageCount=15&id=124253 .
38. ГОСТ Р ИСО/МЭК 12207-2010 Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств [Электронный ресурс]. - URL: http://protect.gost.ru/v.aspx?control=8&baseC=-1&page=0&month=-1&year=-1&search=&RegNum=1&DocOnPageCount=15&id=169094 .
39. ГОСТ 34.601-90 Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания [Электронный ресурс]. - URL: http://protect.gost.ru/v.aspx?control=8&baseC=-1&page=0&month=-1&year=-1&search=&RegNum=1&DocOnPageCount=15&id=129655 .
40. ГОСТ 19.701-90 Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения [Электронный ресурс]. - URL: http://protect.gost.ru/v.aspx?control=8&baseC=-

1&page=0&month=-1&year=-1&search=&RegNum=1&DocOnPageCount=15&id=129742 .
41. Dennis M. Ritchie and Ken Thompson "The UNIX Timesharing Operating System". Bell Laboratories [Электронный ресурс]. - URL: https://pdos.csail.mit.edu/6.828/2005/readings/ritchie74unix.pdf
42. UART: RS232 [Электронный ресурс]. - URL: https://www.camiresearch.com/Data_Com_Basics/RS232_standard.html
43. MD6 Message Digest 6 [Электронный ресурс]. - URL: https://ru.wikipedia.org/wiki/MD6 .
44. Циклический избыточный код (CRC8, CRC16, CRC32) [Электронный ресурс]. - URL: https://ru.wikipedia.org/wiki/Циклический_избыточный_код .
45. UML 2.4.1 принят в качестве международного стандарта ISO/IEC 19505-1, 19505-2 [Электронный ресурс]. - URL: https://www.omg.org/spec/UML/ .
46. Альт Линукс [Электронный ресурс]. - URL: http://altlinux.ru .
47. Форум сообщества Альт Линукс [Электронный ресурс]. - URL: http://forum.altlinux.ru .
48. Язык C/C++ (справочник на английском языке с примерами) [Электронный ресурс]. - URL: http://cplusplus.com .
49. Общение и трудоустройство сообщества программистов. [Электронный ресурс]. - URL: stackoverflow.com .