

ИСКУССТВО ОТЛАДКИ

ИСКУССТВО ОТЛАДКА

с GDB, DDD и Eclipse

Норман Мэтлофф и Питер Джей Зальцман



Сан-Франциско

ИСКУССТВО ОТЛАДКИ С GDB, DDD И ECLIPSE. Авторские права © 2008 принадлежат Норману Мэтлоффу и Питеру Джоу Зальцману.

Все права защищены. Никакая часть этой работы не может быть воспроизведена или передана в любой форме или любыми средствами, электронными или механическими, включая фотокопирование, запись или любую систему хранения или поиска информации, без предварительного письменного разрешения владельца авторских прав и издателя.

12 11 10 09 08 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-002-X

ISBN-13: 978-1-59327-174-9

Издатель: Уильям Поллок

Редактор-постановщик: Меган Данчак

Дизайн обложки и интерьера: Octopod Studios

Редактор по развитию: Тайлер Ортман

Технический рецензент: Дэниел Яковович

Редактор: Нил Чинг

Композитор: Райли Хоффман

Корректор: Рейчел Кай

Индексатор: Фред Браун, Allegro Technical Indexing

Для получения информации о дистрибуторах книг или переводах обращайтесь напрямую в компанию No Starch Press,
Inc.: No Starch Press, Inc.

555 De Haro Street, Suite 250, Сан-Франциско, Калифорния 94107,

телефон: 415.863.9900; факс: 415.863.9950; info@nostarch.com; www.nostarch.com

Каталогизация данных публикаций Библиотеки Конгресса

Мэтлофф, Норман С.

Искусство отладки с помощью GDB, DDD и Eclipse / Норман Мэтлофф и Р
Зальцман.

СМ.

ISBN-13: 978-1-59327-002-5

ISBN-10: 1-59327-002-X

1. Отладка в информатике. 2. Компьютерное программное обеспечение - Контроль качества. I.

Зальцман, П. Дж. II. Название.

QA76.9.D43M35 2008

005.1'4-dc22

2003017566

No Starch Press и логотип No Starch Press являются зарегистрированными товарными знаками No Starch Press, Inc. Другие названия продуктов и компаний, упомянутые здесь, могут быть товарными знаками их соответствующих владельцев. Вместо того, чтобы использовать символ товарного знака при каждом упоминании товарного знака, мы используем эти названия только в редакционной манере и в интересах владельца товарного знака, без намерения нарушить товарный знак.

Информация в этой книге распространяется на условиях «как есть», без гарантии. Хотя при подготовке этой работы были приняты все меры предосторожности, ни авторы, ни No Starch Press, Inc. не несут никакой ответственности перед любым лицом или организацией в отношении любых потерь или ущерба, причиненных или предположительно причиненных прямо или косвенно содержащейся в ней информацией.

КРАТКОЕ СОДЕРЖАНИЕ

Предисловие	xi
Глава 1: Некоторые предварительные сведения для новичков и профессионалов	1
Глава 2 : Остановка , чтобы осмотреться	117
5: Отладка в контексте с несколькими действиями	185
Глава 7 : Другие инструменты.	235
Индекс.	

ПОДРОБНОЕ СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

xi

1

НЕКОТОРЫЕ ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ ДЛЯ НОВИЧКОВ И ПРОФИ	1
1.1 Инструменты отладки, используемые в этой книге 1.2	1 2
Фокус на языке программирования 1.3 Принципы отладки	2
1.3.1 Суть отладки: принцип подтверждения 1.3.2 Какую ценность представляет собой инструмент отладки для принципа подтверждения? 3 1.3.3 Другие принципы отладки..... 4	2
1.4 Текстовые и графические инструменты отладки и компромисс между ними... 5	
1.4.1 Краткое сравнение интерфейсов 5 1.4.2 Компромиссы..... 12	
1.5 Основные операции отладчика	14
1.5.1 Пошаговое выполнение 1.5.2 исходного кода. Проверка переменных 1.5.3 Выпуск «Бюллетеня всех точек» для изменений переменной 1.5.4 Перемещение вверх и вниз по стеку вызовов	14 15 17 17
1.6 Онлайн -помощь..... 19	
1.7 Вводный сеанс отладки 19	
Подход GDB Тот же сеанс в DDD 1.7.2 36	1.7.1 22 1.7.2 36
Сессия в Eclipse	1.7.3 38
1.8 Использование файлов запуска 43.....	

2

ОСТАНДИЛИВАЮСЬ, ЧТОБЫ ОЗНАКОМИТЬСЯ ВОКРУГ	47
2.1 Механизмы паузы 2.2 Обзор точек останова	47 48
2.3 Отслеживание точек останова	49
2.3.1 Списки точек останова 2.3.2 в GDB Списки точек 2.3.3 останова в DDD Списки точек останова в Eclipse	49 50 51
2.4 Установка точек останова	51
2.4.1 Установка точек останова 2.4.2 в GDB Установка точек 2.4.3 останова в DDD Установка точек останова в Eclipse	51 55 56
2.5 Расширенный пример GDB	56

2.6 Сохранение точек останова	59
2.7 Удаление и отключение точек останова	60
2.7.1 Удаление точек останова в GDB 61	
2.7.2 Отключение точек останова в GDB 62	
2.7.3 Удаление и отключение точек останова в DDD 62	
2.7.4 Удаление и отключение точек останова в Eclipse	63
2.7.5 «Перемещение» точек останова в DDD	64
2.7.6 Отмена/повтор действий точки останова в DDD 66	
2.8 Подробнее о просмотре атрибутов точек останова 67.....	
2.8.1 ГБД..... 67	
2.8.2 ДДД..... 69	
2.8.3 Затмение	69
2.9 Возобновление исполнения 69.....	
2.9.1 В ГБД 70	
2.9.2 В ДДД 78	
В Затмении	2.9.3 79
2.10 Условные точки останова	79
2.10.1 ГБД..... 80	
2.10.2 ДДД..... 83	
2.10.3 Затмение	84
2.11 Списки команд точек останова	85
2.12 Точки наблюдения	89
2.12.1 Установка точек наблюдения..... 90	
2.12.2 Выражения	93
3	
ПРОВЕРКА И НАСТРОЙКА ПЕРЕМЕННЫХ	95
3.1 Наш основной пример кода..... 95	
3.2 Расширенная проверка и настройка переменных	98
3.2.1 Проверка в GDB 98	
3.2.2 Инспекция в DDD..... 102	
3.2.3 Проверка в Eclipse..... 103	
3.2.4 Проверка динамических массивов	104
3.2.5 Что насчет C++? 109	
3.2.6 Мониторинг локальных переменных	112
3.2.7 Прямое исследование памяти	112
3.2.8 Расширенные параметры печати и отображения	112
3.3 Установка переменных из GDB/DDD/Eclipse	113
3.4 Собственные переменные GDB	113
3.4.1 Использование истории значений	114
3.4.2 Удобные переменные	114
4	
КОГДА ПРОГРАММА ВЫХОДИТ ИЗ СТРОЯ	117
4.1 Вводные материалы: Управление памятью	118

4.1.1 Почему программа аварийно завершает работу?	118
4.1.2 Размещение программы в памяти 4.1.3 Понятие страниц	118 121
4.1.4 Подробная информация о роли таблицы страниц 4.1.5 Незначительная ошибка доступа <small>Нет</small> к памяти может вызвать ошибку сегмента 4.1.6 Ошибки сегмента и сигналы Unix	122 124 125
4.1.7 Другие типы исключений	128
4.2 Основные файлы..... 129	
4.2.1 Как создаются основные файлы 4.2.2	129
Ваша оболочка может подавлять создание файла ядра	130
4.3 Расширенный пример	131
4.3.1 Первая ошибка	135
4.3.2 Не покидайте GDB во время сеанса отладки	137
4.3.3 Вторая и третья ошибки	137
4.3.4 Четвертая ошибка	139
Пятый и шестой баги	4.3.5
	141
 5 ОТЛАДКА В КОНТЕКСТЕ МНОЖЕСТВА ДЕЯТЕЛЬНОСТИ	145
5.1 Отладка сетевых программ клиент/сервер	145
5.2 Отладка потокового кода..... 151	
5.2.1 Обзор процессов и потоков..... 151	
Базовый пример	5.2.2 153
5.2.3 Вариант	159
5.2.4 Обзор команд потоков GDB	161
5.2.5 Команды потоков в DDD..... 161	
5.2.6 Команды потоков в Eclipse	161
5.3 Отладка параллельных приложений	163
5.3.1 Системы передачи сообщений	164
5.3.2 Системы с общей памятью	170
5.4 Расширенный пример	171
5.4.1 Обзор OpenMP	171
5.4.2 Пример программы OpenMP	172
 6 СПЕЦИАЛЬНЫЕ ТЕМЫ	185
6.1 Что делать, если он даже не компилируется или не	185
Фантомные номера строк в сообщениях об ошибках синтаксиса	185
загружается? <small>Отсутствующие</small> библиотеки..... 190	
6.2 Отладка программ с графическим интерфейсом пользователя	194
6.2.1 Отладка программ Curses	194
 7 ДРУГИЕ ИНСТРУМЕНТЫ	205

7.1 Эффективное использование текстового редактора	206
Подсветка синтаксиса	206	
редактора 7.1.1 7.1.2	208	
Соответствующие скобки 7.1.3 Vim и Makefiles	209	
7.1.4 Makefiles и предупреждения компилятора	210	
7.1.5 Заключительные мысли о текстовом редакторе как IDE 211		
7.2 Эффективное использование компилятора	212	
7.3 Отчеты об ошибках в C 213		
7.3.1 Использование ошибки 213		
7.4 Лучшая жизнь с strace и ltrace 217		
7.5 Статические средства проверки кода: lint и его друзья.....	219	
7.5.1 Как использовать шину	221	
7.5.2 Последние слова.....	221	
7.6 Отладка динамически выделяемой памяти	221	
Стратегии обнаружения проблем DAM 7.6.1	224	
Электрический забор 7.6.2	225	
Отладка проблем DAM с помощью инструментов библиотеки GNU C	228	
8 ИСПОЛЬЗОВАНИЕ GDB/DDD/ECLIPSE ДЛЯ ДРУГИХ ЯЗЫКОВ		235
8.1 Ява.....	236	
8.1.1 Прямое использование GDB для отладки Java.....	238	
8.1.2 Использование DDD с GDB для отладки Java.....	240	
8.1.3 Использование DDD в качестве графического интерфейса для JDB	241	
8.1.4 Отладка Java в Eclipse	241	
8.2 Perl 242		
8.2.1 Отладка Perl через DDD 244		
8.2.2 Отладка Perl в Eclipse.....	246	
8.3 Питон	247	
8.3.1 Отладка Python в DDD	249	
8.3.2 Отладка Python в Eclipse	250	
8.4 Отладка кода SWIG 8.5 Язык ассемблера	255	
ИНДЕКС		259

ПРЕДИСЛОВИЕ

«Эй, эта штука действительно работает!» Так сказал один из наших студентов, Эндрю, после того, как он впервые серьезно использовал инструмент отладки. Он узнал об инструментах отладки три года назад на своих курсах программирования для первокурсников, но он отверг их как просто

что-то, что нужно выучить для выпускного экзамена. Теперь, когда Эндрю был студентом четвертого курса, его профессор настоятельно рекомендовал ему прекратить использовать операторы печати для отладки и использовать формальные инструменты отладки. К его радости и удивлению, он обнаружил, что может значительно сократить время отладки времени, используя соответствующие инструменты.

Среди студентов и работающих программистов много «Эндрю», и мы надеемся, что эта книга вызовет у вас «Эндрюсское» озарение. Для них. Но еще больше мы надеемся охватить многих людей, которые используют инструменты отладки, но не уверены, что можно сделать в особых ситуациях и кто хотел бы узнать больше об инструментах отладки и философии за ними.

Как отметил редактор этой книги, в некоторых сообществах существует много знаний об отладке, представляющих собой своего рода фольклор, но они не описаны в книгах.

Ну, эта книга изменит это. Мы рассмотрим такие вопросы, как:

- Как отлаживать код потоков? • Почему точки останова иногда оказываются в немного других местах, чем те, где вы их установили?
- Почему команда GDB until иногда переходит к «удивительному» место?
- Какие интересные трюки можно сделать с DDD и Eclipse? • В нынешнюю эпоху графического интерфейса пользователя есть ли какие-либо текстовые приложения, такие как GDB? ценить?
- Почему не возникло ошибки сегментации, когда ваш ошибочный код вышел за пределы массива?
 - Почему одна из наших структур данных в качестве примера называется nsp? (Извините, это внутренняя шутка нашего издателя.)

Эта книга не является ни прославленным руководством пользователя, ни абстрактным трактатом по когнитивной теории процесса отладки. Вместо этого она представляет собой нечто среднее между этими двумя жанрами. С одной стороны, мы действительно даем информацию о том, «как» делать определенные команды в GDB, DDD и Eclipse; но с другой стороны, мы излагаем и часто используем некоторые общие принципы для процесса отладки.

Мы выбрали GDB, DDD и Eclipse в качестве иллюстративных инструментов из-за их популярности в сообществах Linux/open-source. Наши примеры немного отдают предпочтение GDB, не только потому, что его текстовая природа делает его более компактным для представления на странице, но и потому, что, как упоминалось выше, мы обнаруживаем, что текстовые команды по-прежнему играют важную роль в процессе отладки.

Eclipse стал довольно широко использоваться для гораздо большего, чем просто отладочная роль, которую мы рассматриваем здесь, и он действительно предоставляет привлекательный, универсальный инструмент для отладки. С другой стороны, DDD имеет гораздо меньший след и включает в себя некоторые мощные функции, которых нет в Eclipse.

Глава 1 «Некоторые подготовительные мероприятия для новичков и профессионалов» представляет собой обзор. Многие опытные программисты могут поддаться искушению пропустить его, но мы настоятельно рекомендуем им прочитать его, поскольку мы излагаем ряд простых, но эффективных общих рекомендаций, которые мы рекомендуем для процесса отладки.

Затем Глава 2, «Остановка, чтобы осмотреться», посвящена рабочей лошадке отладки, точки останова, обсуждение всех тонкостей — установка, удаление и отключение точек останова; переход от одной точки останова к другой; просмотр подробной информации о точках останова и т. д.

Когда вы достигаете точки останова, что происходит? Глава 3, «Проверка и установка переменных», рассматривает этот вопрос. Наш работающий пример здесь касается кода, который проходит по дереву. Ключевым моментом является удобное отображение содержимого узла в дереве, когда мы достигаем точки останова. Здесь GDB действительно блистает, предоставляя некоторые очень гибкие функции, которые позволяют вам эффективно отображать интересующую информацию каждый раз, когда программа останавливается. И мы

представляют собой особенно приятную функцию DDD для графического отображения деревьев и другие связанные структуры данных.

Глава 4, «Когда программа дает сбой», охватывает ужасные ошибки времени выполнения, возникающие из-за ошибок сегментации. Сначала мы представляем материал о том, что такое происходящие на нижних уровнях, включая выделение памяти для программы и совместные роли оборудования и операционной системы. Читатели с хорошими знаниями систем могут бегло просмотреть этот материал, но мы верим, что многие другие выиграют, приобретя этот фонд. Мы тогда перейти к основным файлам — как они создаются, как их использовать для выполнения «постобработки» mortems и т. д. Мы завершаем главу развернутым примером сеанс отладки, в котором несколько ошибок приводят к сбоям сегментации.

В качестве названия мы выбрали «Отладка в контексте множественных действий». В главе 5 подчеркивается, что мы рассматриваем не только параллельное программирование но и сетевой код. Клиент-серверное сетевое программирование считается параллельная обработка, даже наши инструменты используются параллельно, например, два окна, в которых мы используем GDB, одно для клиента, одно для сервера.

Поскольку сетевой код включает системные вызовы, мы дополняем нашу отладку инструменты с переменной C/C++ errno и командой Linux strace .

Следующая часть главы 5 посвящена программированию потоков. Здесь мы снова начинаем с обзора инфраструктуры: разделения времени, процессов и потоков, Условия гонки и т. д. Мы представляем технические детали работы с потоки в GDB, DDD и Eclipse и снова обсудим некоторые общие принципы, которые следует иметь в виду, такие как случайность времени, в котором потоки Происходит переключение контекста. Заключительная часть главы 5 посвящена параллельному программированию с популярными пакетами MPI и OpenMP. Мы заканчиваем расширенный пример в контексте OpenMP.

Глава 6, «Специальные темы», охватывает некоторые важные разные темы. Инструмент отладки не поможет вам, если ваш код даже не компилируется, поэтому мы обсуждаем некоторые подходы к решению этой проблемы. Затем мы решаем проблему сбоя при связывании из-за отсутствия библиотек; мы снова почувствовали, что это полезно здесь, чтобы дать немного «теории» — типы библиотек и как они связаны с ваш основной код, например. А как насчет отладки программ с графическим интерфейсом? Для простоты мы придерживаемся здесь настройки «полу-GUI», то есть программирования curses, и показываем, как заставить GDB, DDD и Eclipse взаимодействовать с событиями. в окне ваших проклятий.

Как отмечалось ранее, процесс отладки можно значительно улучшить за счет использования дополнительных инструментов, некоторые из которых мы представляем в Главе 7, «Другие инструменты». У нас есть дополнительное освещение errno и strace, некоторые материалы по lint и советы по эффективному использованию текстового редактора.

Хотя книга посвящена C/C++, в главе 8 «Использование GDB/DDD/Eclipse для других языков» рассматриваются и другие языки. работа с Java, Python, Perl и языком ассемблера.

Приносим извинения, если мы каким-то образом упустили из виду любимую тему отладки читателей, но мы охватили материал, который сочли полезным в наше собственное программирование.

Мы очень благодарны многим сотрудникам No Starch Press. кто помогал нам в этом проекте на протяжении его долгого времени. Мы особенно благодарим

основатель и редактор фирмы Билл Поллок. Он верил в этот необычный проект с самого начала и был на удивление терпим к нашим многочисленным задержкам.

Дэниел Яковиц проделал поистине блестящую работу по рецензированию рукописи, предоставив множество бесценных советов. Нил Чинг, якобы нанятый для редактирования, на самом деле оказался «звонарем» со степенью в области компьютерных наук! Он поднял ряд важных моментов, касающихся ясности наших технических обсуждений. Качество книги значительно улучшилось благодаря отзывам, которые мы получили от Дэниела и Нила. Конечно, следует сделать обычную оговорку, что все ошибки являются нашими собственными.

Норм говорит: Я хочу сказать «Сие, сие» и «Тода рабах» моей жене Гамис и дочери Лоре, двум удивительным людям, с которыми мне повезло быть родственником. Их подход к решению проблем, искрометный юмор и *joie de vivre* пронизывают эту книгу, несмотря на то, что они не прочитали ни слова из нее. Я также благодарю многих студентов, которых я обучал на протяжении многих лет, которые учат меня так же, как я учу их, и которые заставляют меня чувствовать, что я выбрал правильную профессию в конце концов. Я всегда стремился «изменить мир», и надеюсь, что эта книга сделает это хоть немного.

Комментарии Пита: Я благодарю Николь Карлсон, Марка Кима и Ронду Зальцман за то, что они потратили много часов на чтение глав и внесение исправлений и предложений, просто потому, что вы сейчас читаете. Я также хотел бы поблагодарить людей из Linux Users Group в Дэвисе, которые отвечали на мои вопросы на протяжении многих лет. Знакомство с вами сделало меня умнее. Тодах обращается к Эвелин, которая улучшила мою жизнь во всех отношениях. Особого упоминания заслуживает Джорди («J-Train» из Сан-Франциско), который бескорыстно использовал свой собственный кошачий вес, чтобы страницы не унесло ветром, всегда держал мое сиденье в тепле и следил за тем, чтобы комната никогда не пустовала.

Тебя очень не хватает каждый день. Мурлыкай, малыш. Привет, мамочка!

Посмотрите, что я сделал!

Норм Мэтлофф и Пит Зальцман 9 июня

2008 г.

1

НЕКОТОРЫЕ ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ ДЛЯ НОВИЧКИ И ПРОФИ



Некоторые люди, особенно профессионалы, могут поддаться искушению пропустить эту главу. Мы предлагаем, однако, всем хотя бы бегло просмотреть ее. Многие профессионалы найдут некоторые материалы новыми для себя, и в любом случае важно, чтобы все читатели были знакомы с представленным здесь материалом, который будет использоваться на протяжении всей оставшейся части книги. Новичкам, конечно, следует внимательно прочитать эту главу.

В первых нескольких разделах этой главы мы дадим обзор процесса отладки и роли инструментов отладки, а затем рассмотрим расширенный пример в разделе 1.7.

1.1 Инструменты отладки, используемые в этой книге

В этой книге мы излагаем основные принципы отладки, иллюстрируя их в контексте следующих инструментов отладки:

ГБД

Наиболее часто используемый инструмент отладки среди программистов Unix. это GDB, отладчик проекта GNU, разработанный Ричардом Столлманом, видный лидер движения за программное обеспечение с открытым исходным кодом, который сыграл ключевую роль в развитии Linux.

Большинство систем Linux должны иметь предустановленный GDB. Если это не так, вы необходимо загрузить пакет компилятора GCC.

ДДД

В связи с недавней популярностью графических пользовательских интерфейсов (GUI) Разработано несколько отладчиков на основе графического интерфейса, работающих под Unix. Большинство из них — это графические интерфейсы к GDB: пользователь вводит команды через графический интерфейс, который в свою очередь передает их в GDB. Один из них — это DDD, отладчик отображения данных.

Если в вашей системе еще не установлен DDD, вы можете...
загрузить его. Например, в системах Fedora Linux команда

нам установить ддд

позаботится обо всем процессе за вас. В Ubuntu Linux аналогичный можно использовать команду apt-get .

Затмение

Некоторые читатели могут использовать интегрированные среды разработки (IDE). IDE — это больше, чем просто инструмент отладки; он объединяет редактор, сборку инструмент, отладчик и другие средства разработки в одном пакете. В этом В нашей книге в качестве примера IDE используется очень популярная система Eclipse. Как и в случае с DDD, Eclipse работает поверх GDB или какого-либо другого отладчика.

Вы можете установить Eclipse через yum или apt-get, как указано выше, или просто зайти в...
загрузите .zip- файл и распакуйте его в подходящий каталог, например /usr/local.

В этой книге мы используем Eclipse версии 3.3.

1.2 Фокус на языке программирования

В этой книге мы в первую очередь рассматриваем программирование на языке C/C++, и большинство из наши примеры будут в этом контексте. Однако в Главе 8 мы обсудим другие языки.

1.3 Принципы отладки

Несмотря на то, что отладка — это скорее искусство, чем наука, существуют определенные принципы, которые направляют его практику. Мы обсудим некоторые из них в этом разделе.

По крайней мере одно из наших правил, Основополагающий Принцип Подтверждения, заключается в следующем: довольно формальный характер.

1.3.1 Из Сущность Отладка: The Принцип Подтверждение

Следующее правило является сутью отладки:

Основной принцип подтверждения

Исправление неисправной программы — это процесс последовательного подтверждения того, что многие вещи, которые вы считаете правдой о коде, на самом деле правдой . Когда вы обнаружили, что одно из ваших предположений не верно, вы нашли подсказку к местоположению (если не к точной природе) ошибки.

Другими словами это можно выразить так:

Сюрпризы — это хорошо!

Когда одна из вещей, которые вы считаете правдой о программе, не выполняется подтверждаю, вы удивлены. Но это хороший сюрприз, потому что это открытие может привести вас к местоположению ошибки.

1.3.2 Какова ценность этого инструмента отладки для принципа подтверждения?

Классический метод отладки — простое добавление кода трассировки в программу для вывода значений переменных во время выполнения программы с помощью printf() или Например, операторы cout . Вы можете спросить: «Разве этого недостаточно? Зачем использовать инструмент отладки, такой как GDB, DDD или Eclipse?»

Прежде всего, этот подход требует постоянного цикла стратегического добавления трассировка кода, перекомпиляция программы, запуск программы и анализ вывод кода трассировки, удаление кода трассировки после исправления ошибки, и повторение этих шагов для каждой новой обнаруженной ошибки. Это очень отнимает много времени и утомляет. Самое главное, эти действия отвлекают отвлекает вас от реальной задачи и снижает вашу способность сосредоточиться на процессе рассуждения, необходимом для поиска ошибки.

Напротив, с графическими инструментами отладки, такими как DDD и Eclipse, все что вам нужно сделать, чтобы проверить значение переменной, это переместить мышь указатель мыши на экземпляр этой переменной в отображении кода, и вы показал его текущую стоимость. Зачем утомлять себя еще больше, чем нужно, больше, чем необходимо, во время ночной сессии отладки, выполняя это с использованием операторов printf() ? Сделайте себе одолжение и уменьшите количество времени, которое вам придется потратить, и скуки, которую вам придется терпеть, используя инструмент отладки.

Вы также получаете гораздо больше от инструмента отладки, чем просто возможность смотреть переменные. Во многих ситуациях отладчик может сказать вам приблизительное местоположение ошибки. Предположим, например, что ваша программа зависает или вылетает с ошибкой сегментации, то есть с ошибкой доступа к памяти. Как вы увидите в нашем Пример сеанса отладки далее в этой главе. GDB/DDD/Eclipse может немедленно указать вам место ошибки сегмента, которое обычно находится на или около местонахождение ошибки.

Аналогично, отладчик позволяет вам устанавливать контрольные точки , которые могут сообщить вам, в какой момент момент во время выполнения программы значение определенной переменной достигает подозрительное значение или диапазон. Этую информацию может быть трудно вывести, глядя на вывод вызовов printf().

1.3.3 Другой Отладка Принципы

Начать с малого

В начале процесса отладки вы должны запустить вашу программу на простых, легких случаях. Это может не выявить все ваши ошибки, но это вероятно, раскроет некоторые из них. Если, например, ваш код состоит из большой цикл, легче всего обнаружить ошибки, которые возникают на первом или вторая итерация.

Используйте подход «сверху вниз»

Вы, вероятно, знаете об использовании нисходящего или модульного подхода к написанию кода: ваша основная программа не должна быть слишком длинной, и она должна состоять в основном из вызовов функций, которые выполняют существенную работу. Если один из них функции длинные, вам следует рассмотреть возможность разбить их, в свою очередь, на меньшие модули.

Вам следует не только писать код сверху вниз, но и также отлаживайте код сверху вниз.

Например, предположим, что ваша программа использует функцию `f()`. Когда вы пройдитесь по коду с помощью инструмента отладки и обнаружите вызов `f()`, отладчик предоставит вам выбор, где произойдет следующая пауза в выполнении — либо в первой строке функции, которая вот-вот завершится, быть вызвана или в операторе, следующем за вызовом функции. Во многих случаях, последний вариант является лучшим первоначальным выбором: вы выполняете вызов, а затем проверяете значения переменных, которые зависят от результатов вызова, чтобы увидеть, правильно ли работает функция. Если да, то вы избежите отнимающих много времени и ненужных усилий, связанных с перешагиванием через код внутри функции, который не вел себя неправильно (в данном случае).

Используйте инструмент отладки, чтобы определить место ошибки сегментации.

Самый первый шаг, который вы должны предпринять при возникновении ошибки сегмента, — это запустить вашу программу в отладчике и воспроизведите ошибку сегмента. Отладчик укажет вам строку кода, в которой произошла ошибка. Вы затем можно получить дополнительную полезную информацию, вызвав отладчик средство обратной трассировки, которое отображает последовательность вызовов функций, приводящих к вызову функции, в которой произошла ошибка.

В некоторых случаях может быть сложно воспроизвести ошибку сегмента, но если у вас есть файл ядра, вы все равно можете сделать обратную трассировку, чтобы определить ситуацию, которая привела к ошибке сегмента. Это будет обсуждаться в Главе 4.

Определите местоположение бесконечного цикла, выполнив прерывание

Если вы подозреваете, что ваша программа имеет бесконечный цикл, войдите в отладчик и снова запустите программу, дав ей поработать достаточно долго, чтобы войти в `loop`. Затем используйте команду прерывания отладчика, чтобы приостановить программу, и выполните обратную трассировку, чтобы увидеть, какая точка тела цикла была достигнуто и как программа туда попала. (Программа не была убит; вы можете возобновить казнь, если пожелаете.)

Использовать бинарный поиск

Вы, вероятно, видели бинарный поиск в контексте отсортированных списков. Скажем, для пример, у вас есть массив `x[]` из 500 чисел, расположенных в порядке возрастания.

порядок, и вы хотите определить, куда вставить новый номер, у. Начните со сравнения у с x[250]. Если у окажется меньше этого элемента, то вы сравните его с x[125], но если у больше x[250], то следующее сравнение будет с x[375]. В последнем случае, если у меньше x[375], затем вы сравниваете его с x[312], который находится на полпути между x[250] и x[375] и т. д. Вы бы продолжали сокращать свой поиск пробелы уменьшаются вдвое при каждой итерации, что позволяет быстро найти точку вставки.

Этот принцип можно применять и при отладке. Предположим, вы знаете, что значение, хранящееся в определенной переменной, становится плохим в течение первых 1000 итераций цикла. Один из способов, который может вам помочь отследить итерацию, где значение впервые становится плохим, можно с помощью точки наблюдения, продвинутой техники, которую мы обсудим в разделе 1.5.3. Другой подход заключается в использовании бинарного поиска, в данном случае по времени, а не в пространстве. Сначала вы проверяете значение переменной на 500-й итерации; если на этом этапе все еще в порядке, затем вы проверяете значение на 750-м итерация и т. д.

В качестве другого примера предположим, что один из исходных файлов в вашей программе даже не компилируется. Стока кода, указанная в сообщении компилятора, генерированном синтаксической ошибкой, иногда находится далеко от фактического местоположения ошибки, и поэтому у вас могут возникнуть проблемы с определением этого местоположения. Двоичный поиск может помочь здесь: вы удаляете (или закомментируете) один половину кода в единице компиляции, перекомпилируйте оставшийся код, и посмотрите, сохранился ли сообщение об ошибке. Если это так, то ошибка в этом второй половине; если сообщение не появляется, то ошибка в половине что вы удалили. Как только вы определите, какая половина кода содержит насекомое, вы далее ограничиваете насекомое половиной этой части и держите пока не обнаружите проблему. Конечно, вам следует сделать копию исходного кода перед началом этого процесса или, что еще лучше, используйте свой функция отмены текстового редактора. Советы по эффективному использованию см. в Главе 7. редактор во время программирования.

1.4 Текстовые и графические инструменты отладки и компромисс между ними

Графические интерфейсы, обсуждаемые в этой книге, DDD и Eclipse, служат в качестве интерфейсов для GDB для C и C++ и для других отладчиков. В то время как GUI имеют привлекательный вид и может быть более удобным, чем текстовый GDB, наша точка зрения В этой книге будут рассмотрены текстовые и графические отладчики (включая IDE) все они полезны в разных контекстах.

1.4.1 Краткий Сравнение Интерфейсы

Чтобы быстро получить представление о различиях между текстовыми и графическими инструментами отладки, давайте рассмотрим ситуацию, которую мы будем использовать в качестве рабочего примера. этой главе. Программа в примере — `insert_sort`. Она скомпилирована из исходный файл `ins.c`, и он выполняет сортировку вставкой.

1.4.1.1 GDB: простой текст

Чтобы начать сеанс отладки этой программы с помощью GDB, введите:

```
$ gdb вставка_сортировки
```

в командной строке Unix, после чего GDB предложит вам ввести команды, отобразив приглашение:

```
(гдб)
```

1.4.1.2 DDD: инструмент отладки графического

интерфейса пользователя. Используя DDD, вы можете начать сеанс отладки, введя

```
$ дdd вставка_сортировка
```

в командной строке Unix. Появится окно DDD, после чего вы сможете отправлять команды через GUI.

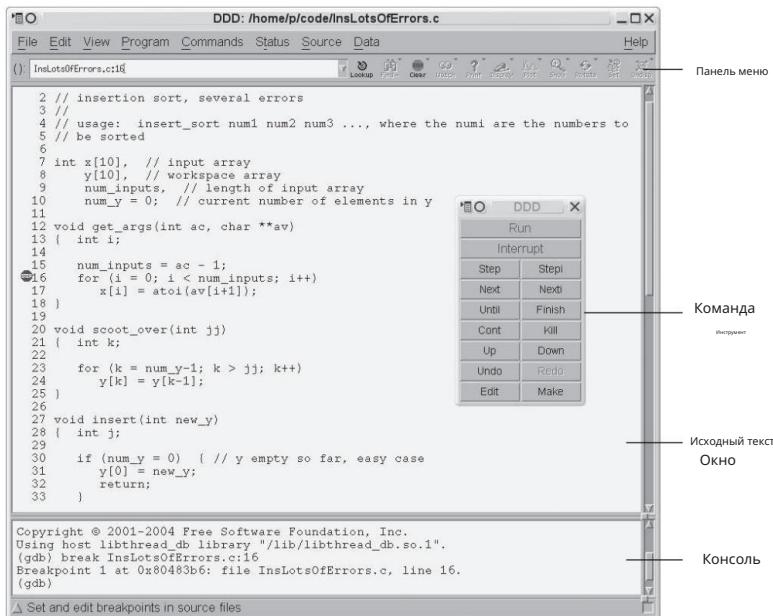
Типичный вид окна DDD показан на рисунке 1-1. Как вы видите ли, окно DDD отображает информацию в различных подокнах:

- Окно исходного текста отображает ваш исходный код. DDD начинает свой дисплей в вашей функции `main()`, но вы, конечно, можете перейти к другим частям исходного файла, используя полосы прокрутки на правом краю окна. • Панель меню представляет различные категории меню, включая Файл, Правка, и Просмотр.
- В Command Tool перечислены наиболее распространенные команды DDD (такие как Run, Interrupt, Step и Next), так что вы можете быстро получить к ним доступ. • Консоль: помните, что DDD — это просто графический интерфейс пользователя для GDB (и в другие отладчики). DDD преобразует выбор, сделанный с помощью мыши, в соответствующие команды GDB. Эти команды и их вывод отображаются в Консоли. Кроме того, вы можете отправлять команды в GDB напрямую через Консоль, что является удобной функцией, поскольку не все команды GDB имеют аналоги DDD.
- Окно данных показывает значения переменных, которые вы запросили для постоянного отображения. Это подокно не появится, пока вы не сделаете такой запрос, поэтому оно не отображается на этом рисунке.

Вот краткий пример того, как типичная команда отладки отправляется отладчику в каждом типе пользовательского интерфейса. При отладке `insert_sort` вы можете захотеть приостановить выполнение программы — чтобы установить точку останова — на строке 16 (например) функции `get_args()`. (Вы увидите полный исходный код `insert_sort` в разделе 1.7.) Чтобы организовать это в GDB, вы должны ввести

```
(гдб) перерыв 16
```

в командной строке GDB.

Рисунок 1-1: **ДДД** макет

Полное имя команды — `break`, но GDB допускает сокращения длиной

поскольку здесь нет никакой двусмыслинности, и большинство пользователей GDB напечатали бы здесь `b` 16. Для того, чтобы

Чтобы облегчить понимание для новичков в GDB, мы будем использовать полную команду

Сначала назовите их по именам, а затем, когда команды станут более привычными, перейдите к сокращениям.

Используя DDD, вы должны открыть окно «Исходный текст», щелкнуть на начало строки 16, а затем нажмите значок «Разрыв» в верхней части DDD screen. Вы также можете щелкнуть правой кнопкой мыши в начале строки, а затем выбрать `Set Breakpoint`. Еще один вариант — просто дважды щелкнуть строку код, где угодно слева от начала строки. В любом случае, DDD будет подтвердить выбор, отобразив небольшой знак остановки на этой строке, как показано на рисунке Рисунок 1-2. Таким образом, вы можете сразу увидеть свои контрольные точки.

1.4.1.3 Eclipse: отладчик с графическим интерфейсом и многое другое

Рисунок 1-3 представляет общую среду в Eclipse. В Eclipse терминологии, мы сейчас находимся в перспективе Debug. Eclipse — это общая фреймворк для разработки множества различных видов программного обеспечения. Каждый язык программирования имеет свой собственный подключаемый графический интерфейс — перспективу — в Eclipse. Действительно, может быть несколько конкурирующих перспектив для одного и того же языка. В нашей работе Eclipse в этой книге мы будем использовать перспективу C/C++ для разработки на C/C++, перспективу Pydev для написания программ на Python, и т. д. Также есть перспектива Debug для фактической отладки (с некоторыми особенностями, специфичные для языка), и это то, что вы видите на рисунке.

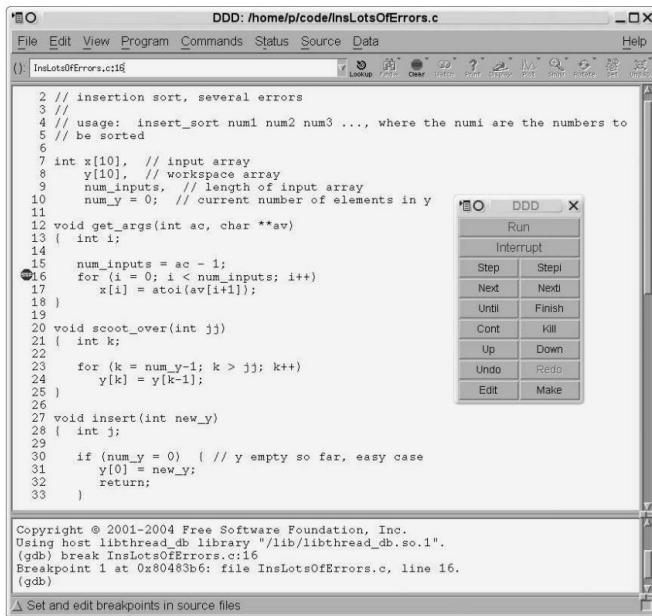


Рисунок 1-2: установлена точка останова

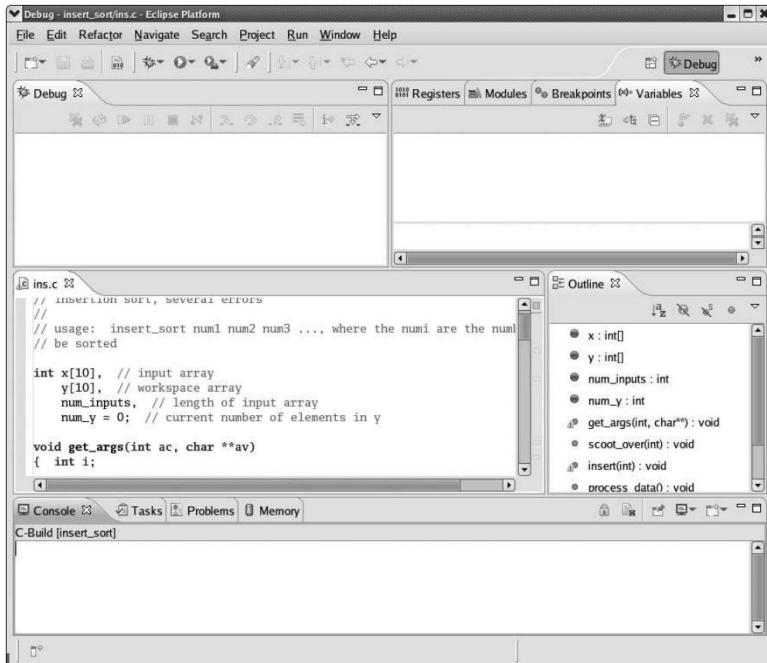


Рисунок 1-3: Окружающая среда Eclipse

Перспектива C/C++ является частью плагина CDT. За кулисами CDT вызывает GDB, аналогично случаю DDD.

Детали этой фигуры в целом похожи на те, что мы описали для DDD выше. Перспектива разбита на вкладки, называемые представлениями. Вы слева можно увидеть вид исходного файла ins.c ; там есть вид Переменные для проверки значений переменных (пока на картинке их нет); есть Консольное представление, функция которого очень похожа на функцию подокна в DDD то же имя и т. д.

Вы можете устанавливать контрольные точки и т. д. визуально, как в DDD. На рисунке 1-4 для примера, линия

для (i = 0; i < num_inputs; i++)

в окне исходного файла есть синий символ в левом поле, символизирующий что там есть точка останова.

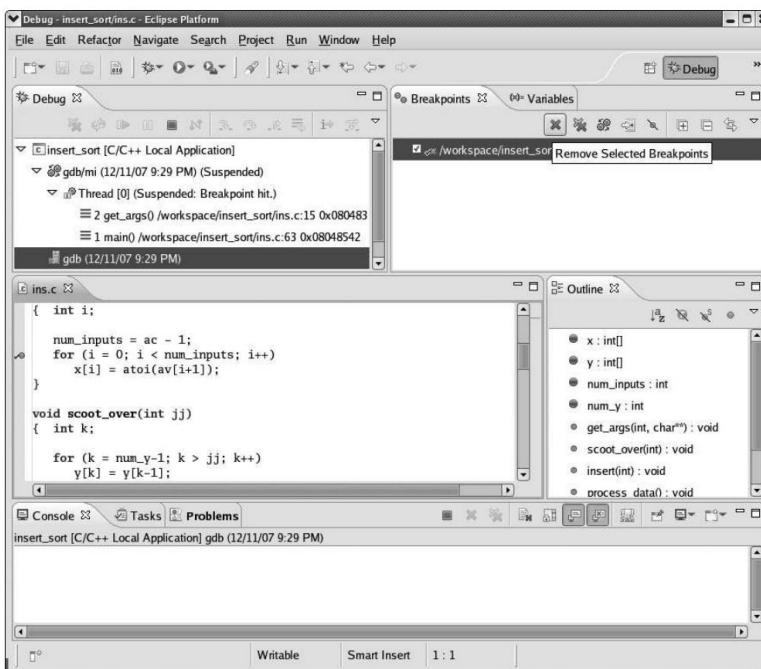


Рисунок 1-4: Удаление ^а точки останова Затмение

1.4.1.4 Eclipse против DDD

В Eclipse также есть некоторые вспомогательные средства, отсутствующие в DDD. Например, около правой стороны обратите внимание на представление Outline, в котором перечислены переменные, функции и т. д. Если вы щелкните запись для вашей функции `scoot_over()`, например, курсор в представлении исходного файла перейдет к этой функции. Более того, если вы временно перейдете из перспективы Debug обратно в перспективу C/C++, где

Вы делаете редактирование и компиляцию для этого проекта (не показано), вид Outline также в вашем распоряжении. Это может быть весьма полезно в больших проектах.

Eclipse также лучше интегрирует процессы редактирования и компиляции. Если у вас есть ошибки компиляции, они четко обозначены в редакторе. Это можно сделать с помощью редактора Vim, который оба автора этой книги предпочитают IDE, но IDE делает это гораздо лучше.

С другой стороны, вы можете видеть, что Eclipse, как и большинство IDE, занимает значительное место на вашем экране (и, конечно же, на страницах этой книги!). Этот вид Outline занимает драгоценное место на экране, независимо от того, используете ли вы его часто или нет. Конечно, вы можете скрыть Outline, нажав на X в его правом углу (и если вы хотите вернуть его обратно, выберите Window | Show Views | Outline), что освобождает немного места, и вы также можете перетаскивать вкладки в разные места в окне Eclipse. Но в целом может быть сложно эффективно использовать пространство экрана в Eclipse.

Помните, что вы всегда можете выполнить команды GDB непосредственно в DDD. Консоль. Таким образом, у вас есть возможность выполнять команды отладки наиболее удобным из доступных способов, иногда через интерфейс DDD, а иногда через командную строку GDB. В различных местах этой книги вы увидите, что есть ряд действий, которые вы можете выполнить с GDB, что может сделать вашу отладочную жизнь намного более удобной.

Напротив, GDB в основном прозрачен для пользователей Eclipse, и хотя старая поговорка «Невежество — это блаженство» может часто применяться, прозрачность означает, что вы теряете легкий доступ к действиям, экономящим труд, которые стали возможны благодаря прямому использованию GDB. На момент написания этой статьи решительный пользователь все еще может напрямую получить доступ к GDB, щелкнув ветку GDB в Debug, а затем используя Console, хотя и без подсказок GDB. Однако эта «недокументированная функция» может не сохраниться в будущих версиях.

1.4.1.5 Преимущества графических интерфейсов
 Интерфейсы графического интерфейса, предоставляемые DDD и Eclipse, визуально более привлекательны, чем интерфейсы GDB. Они также, как правило, более удобны. Например, предположим, что вы больше не хотите, чтобы выполнение останавливалось на строке 16 `get_args()`, то есть вы хотите очистить точку останова. В GDB вы бы очистили точку останова, набрав

```
(gdb) очистить 16
```

Однако, чтобы это сделать, вам нужно запомнить номер строки. Точки останова — непростая задача, если у вас одновременно активны несколько точек останова. Вы можете освежить свою память, используя команду `GDB info break`, чтобы получить список всех точек останова, но это все равно потребует некоторой работы и отвлечет от поиска ошибки.

В DDD ваша задача была бы намного проще: чтобы очистить точку останова, просто щелкните значок «Стоп» на нужной строке, затем щелкните «Очистить», и значок «Стоп» исчезнет, показывая, что точка останова очищена.

В Eclipse вам нужно перейти в представление Breakpoints, выделить точку(ы) останова, которые вы хотите удалить, а затем переместить курсор мыши на серый X, который символизирует операцию Remove Selected Breakpoints (см. рис. 1-4). В качестве альтернативы вы можете щелкнуть правой кнопкой мыши по синему символу точки останова в окне исходного кода и выбрать Toggle Breakpoint.

Одной из задач, в которой графические интерфейсы явно выигрывают, является пошаговое выполнение кода. Гораздо проще и приятнее делать это с помощью DDD или Eclipse, а не GDB, поскольку вы можете наблюдать за своим движением по коду в окне исходного кода GUI. Следующая строка в исходном коде, которая должна быть выполнена, обозначена стрелкой, как показано для DDD на рисунке 1-5. В Eclipse следующая строка выделена зеленым цветом. Таким образом, вы можете одним взглядом определить, где вы находитесь относительно других интересующих вас операторов программы.

1.4.1.6 Преимущества GDB Итак,

GUI имеют много преимуществ перед текстовым GDB. Однако огульное заключение на основе этого примера о том, что GUI лучше GDB, было бы неоправданным.

Молодые программисты, которые выросли, используя GUI для всего, что они делают в сети, естественно, предпочитают GUI GDB, как и многие из их старших коллег. С другой стороны, у GDB есть и некоторые определенные преимущества:

- GDB запускается быстрее, чем DDD, что является большим преимуществом, когда вы просто нужно быстро проверить что-то в коде. Разница во времени запуска еще больше в случае Eclipse.
- В некоторых случаях отладка выполняется удаленно через SSH (или tel-net) соединение, например, с публичного терминала. Если у вас нет настройки X11, GUI вообще не могут быть использованы, и даже с X11 операции обновления экрана GUI могут быть медленными.
- При отладке нескольких программ, работающих совместно друг с другом, например, пары клиент/сервер в сетевой среде, вам необходимо отдельное окно отладки для каждой программы. В Eclipse это немного лучше, чем в DDD, так как Eclipse позволяет вам отлаживать две программы одновременно в одном окне, но это усугубляет проблемы с пространством, упомянутые ранее. Таким образом, небольшой визуальный след, который GDB занимает на экране по сравнению с большим следом GUI, является большим преимуществом.
- Если отлаживаемая вами программа имеет GUI, и вы используете

отладчик на основе GUI, такой как DDD, они могут конфликтовать. События GUI — нажатия клавиш, щелчки мыши и движения мыши — одной программы могут мешать событиям другой, и программа может вести себя по-разному при запуске под отладчиком, чем при запуске независимо. Это может серьезно усложнить поиск ошибок.

Для тех, кто не привык к объему набора текста, необходимому для работы с GDB, По сравнению с удобными операциями мыши в графических интерфейсах, следует отметить, что GDB включает некоторые устройства, экономящие набор текста, которые делают его текстовую природу более приемлемой. Мы уже упоминали ранее, что большинство команд GDB имеют

короткие сокращения, и большинство людей используют их вместо полных форм. Кроме того, комбинации клавиш CTRL-P и CTRL-N позволяют вам прокручивать предыдущие команды и отредактируйте их, если хотите. Просто нажмите клавишу ENTER повторяет последнюю выданную команду (что очень полезно при многократном выполнении следующей команды для пошагового выполнения кода по одной строке за раз) и есть команда define , которая позволяет пользователю определять сокращения и макросы. Подробности этих функций будут представлены в главах 2 и 3.

1.4.1.7 Итог: каждый имеет свою ценность

Мы считаем, что GDB и GUI являются важными инструментами, и эта книга будут представлены примеры GDB, DDD и Eclipse. Мы всегда начнем рассмотрение любой конкретной темы с GDB, поскольку это общее между ними инструменты, а затем показать, как материал распространяется на графические интерфейсы.

1.4.2 Компромиссы

Начиная с версии 6.1, GDB предлагает компромисс между текстовым и графическое взаимодействие с пользователем в виде режима, называемого TUI (Терминал Пользовательский интерфейс). В этом режиме GDB разделяет экран терминала на аналоговые окна исходного текста DDD и консоли; вы можете следить за ходом выполнения выполнение вашей программы в первом случае при выдаче команд GDB в последнее. В качестве альтернативы вы можете использовать другую программу, CGDB, которая предлагает аналогичная функциональность.

1.4.2.1 GDB в режиме TUI

Чтобы запустить GDB в режиме TUI, вы можете указать опцию -tui в командная строка при вызове GDB или нажмите CTRL-XA из GDB, пока в режиме не TUI. Последняя команда также выключает вас из режима TUI, если вы в данный момент находитесь в нем.

В режиме TUI окно GDB разделено на два подокна — одно для Команды GDB и одна для просмотра исходного кода. Предположим, вы запускаете GDB в режиме TUI на insert_sort и затем выполнить пару отладочных команд. Ваш экран GDB может выглядеть следующим образом:

```

11
12     void get_args (int ac, char **av)
13     { int i;
14
15         num_inputs = ac - 1;
* 16         для (i = 0; i < num_inputs; i++)
> 17             X[i] = atoi(av[i+1]);
18     }
19
20     void scoot_over(int jj)
21     { int k;
22
23         для (k = num_y-1; k > jj; k++)

```

Файл: ins.c	Процедура: get_args	Строка: 17	ПК: 0x80484b8
-------------	---------------------	------------	---------------

```
(гdb) перерыв 16
Точка останова 1 по адресу 0x804849f: файл ins.c, строка 16.
(gdb) запустить 12 5 6
Запуск программы: /debug/insert_sort 12 5 6
```

Точка останова 1, get_args (ac=4, av=0xbffff094) в ins.c:16

```
(gdb) следующий
(гdb)
```

Нижнее подокно показывает именно то, что вы бы увидели, если бы использовали...
ing GDB без TUI. Здесь это подокно показывает следующее:

- Мы выполнили команду `break`, чтобы установить точку останова на строке 16 в текущем исходный файл.
- Мы выполнили `run` для запуска программы, передав ей аргументы командной строки 12, 5 и 6, после чего отладчик остановил выполнение на указанная точка останова. (запустится и другие команды GDB будут выполнены)
(объясняется позже.) GDB напоминает нам, что точка останова находится на строке 16 ins.c и сообщает нам, что машинный код для этой исходной строки находится по адресу адрес памяти 0x804849f.
- Мы дали команду `next` для перехода к следующей строке кода, строке 17.

Верхнее подокно предлагает дополнительную визуально полезную информацию.
Здесь TUI показывает нам исходный код, окружающий строку, которая в данный момент обрабатывается.
выполняется, как DDD и Eclipse. Это делает намного проще увидеть
где мы находимся в коде. Точка останова и строка, которая в данный момент выполняется,
обозначены звездочкой и знаком >, соответственно, аналогично
Знак «Стоп» DDD и значки зеленой стрелки.

Мы можем перейти к другим частям кода, используя стрелки вверх и вниз.
клавиши для прокрутки. Если вы не в режиме TUI, вы можете использовать клавиши со стрелками для прокрутки
через предыдущие команды GDB, чтобы изменить или повторить их. В
В режиме TUI клавиши со стрелками служат для прокрутки подокна исходного кода, а
Для прокрутки предыдущих команд GDB используйте сочетания клавиш CTRL-P и CTRL-N.
Также в режиме TUI область кода, отображаемая в подокне исходного кода, может быть изменена
с помощью команды GDB `list`. Это особенно полезно
при работе с несколькими исходными файлами.

Используя режим TUI GDB и его сочетания клавиш, мы можем получить множество
дополнительных функций GUI, не подвергаясь недостаткам GUI. Обратите внимание, однако, что
в некоторых обстоятельствах TUI может вести себя не так, как хотелось бы.
так, как вам хочется, в этом случае вам придется искать обходной путь.

1.4.2.2 CGDB

Другой интерфейс для GDB, который вы, возможно, захотите рассмотреть, — это CGDB, доступный
на <http://cgdb.sourceforge.net/>. CGDB также предлагает компромисс между текстовым

основанный на подходе GUI. Как и GUI, он служит в качестве интерфейса для GDB. Это похоже на концепцию TUI на основе терминала, но с дополнительными соблазнами, что он цветной, и вы можете просматривать исходный код в подокне и устанавливать точки останова прямо там. Он также, кажется, обрабатывает экран обновляется лучше, чем GDB/TUI.

Вот несколько основных команд и соглашений CGDB:

- Нажмите ESC , чтобы перейти из окна команд в окно исходного кода; нажмите я должен вернуться.
- Находясь в исходном окне, перемещайтесь с помощью клавиш со стрелками или клавиши, похожие на vi (j — вниз, k — вверх, / — поиск).
- Следующая строка, которая должна быть выполнена, отмечена стрелкой.
- Чтобы установить точку останова на строке, выделенной курсором, просто нажмите пробел.
- Номера строк точек останова выделены красным цветом.

1.5 Основные операции отладчика

Здесь мы даем обзор основных типов операций, которые может выполнять отладчик. предложения.

1.5.1 Шагать Через Источник Код

Ранее мы видели, что для запуска программы в GDB используется команда run , и что в DDD вы нажимаете Run. В деталях, которые будут представлены позже, вы увидите что Eclipse обрабатывает вещи похожим образом.

Вы также можете сделать так, чтобы выполнение программы приостанавливалось в определенные моменты времени. точек, так что вы можете проверить значения переменных, чтобы получить подсказки о том, где находится ваш баг. Вот некоторые из методов, которые вы можете использовать, чтобы сделать этот:

Точки останова

Как упоминалось ранее, отладочный инструмент приостановит выполнение вашей программы в указанных точках останова. Это делается в GDB с помощью команды break вместе с номером строки; в DDD вы щелкаете правой кнопкой мыши в любом месте в пустом месте в соответствующей строке и выберите Установить точку останова; в Eclipse дважды щелкните на поле слева от строки.

Одношаговый

Следующая команда GDB , которая также упоминалась ранее, сообщает GDB: выполнить следующую строку и затем сделать паузу. Команда step похожа, за исключением того, что при вызовах функций она войдет в функцию, тогда как next приведет к следующей паузе в выполнении, происходящей на строке, следующей за вызовом функции. DDD имеет соответствующие пункты меню Next и Step, в то время как в Eclipse есть значки «Шаг через» и «Шаг внутрь», которые выполняют те же функции.

Возобновить работу

В GDB команда `continue` сообщает отладчику о необходимости возобновить выполнение. и продолжайте до тех пор, пока не будет достигнута точка останова. Есть соответствующее меню элемент в DDD, и в Eclipse для него есть значок «Возобновить».

Временные контрольные точки

В GDB команда `tbreak` похожа на `break`, но устанавливает точку останова, который действует только до первого достижения указанной линии. В DDD это достигается щелчком правой кнопкой мыши в любом месте белого пробел в нужной строке в окне Исходный текст, а затем выбрав Установить временную точку останова. В Eclipse выделите нужную строку в исходное окно, затем щелкните правой кнопкой мыши и выберите «Выполнить по строке». GDB также имеет команды `until` и `finish`, которые создают специальные виды одноразовых точек останова. DDD имеет соответствующие меню Until и Finish элементы в его командном окне, и Eclipse имеет Step Return. Это обсуждается в Главе 2.

Типичная схема отладки для выполнения программы выглядит следующим образом (на примере GDB): после достижения точки останова вы перемещаетесь по код по одной строке за раз или пошагово на некоторое время, с помощью команд GDB `next` и `step`. Это позволяет вам тщательно изучить состояние программы и поведение вблизи точки останова. Когда вы закончите с этим, вы можете сказать отладчику продолжить выполнение программы без остановки до следующего достигнута точка останова с помощью команды `continue`.

1.5.2 Инспекция Переменные

После того, как отладчик приостановит выполнение нашей программы, вы можете выдавать команды для отображения значений переменных программы. Это могут быть локальные переменные, глобальные переменные, элементы массивов и структур C, переменные-члены в C++ классы и т. д. Если переменная имеет неожиданное значение, это обычно является важным ключом к местоположению и природе ошибки. DDD может даже графические массивы, которые могут с первого взгляда выявить подозрительные значения или тенденции, происходящие внутри массива.

Самый простой тип отображения переменной — это просто вывод на экран текущего значения. Например, предположим, что вы установили точку останова на строке 37 функции `insert()` в `ins.c`. (Опять же, полный исходный код приведен в разделе 1.7, но (Подробности пока не должны вас волновать.) Когда вы достигнете этой черты, вы сможете проверьте значение локальной переменной `j` в этой функции. В GDB вы бы используйте команду печати :

(gdb) печать j

В DDD это еще проще: вы просто перемещаете указатель мыши над любым экземпляром `j` в окне исходного текста, а затем значение `j` будет отображаться в течение секунды или двух в маленьком желтом поле, называемом подсказкой значения , рядом с указатель мыши. См. рисунок 1-5, где значение переменной `new_y` изучается. То же самое происходит и с Eclipse, как показано на рисунке 1-6, где мы запрашиваем значение `num_y`.

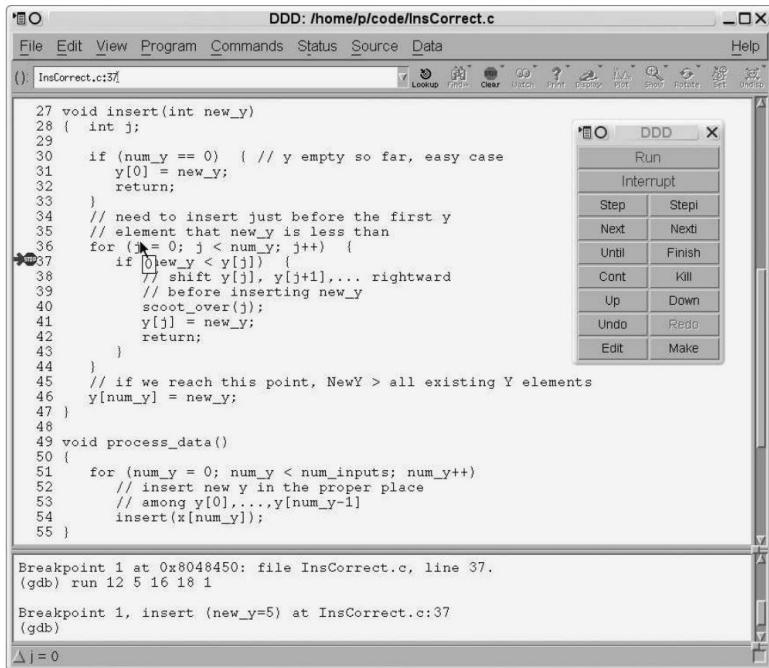


Рисунок 1-5: Инспекция а переменная в ДДД

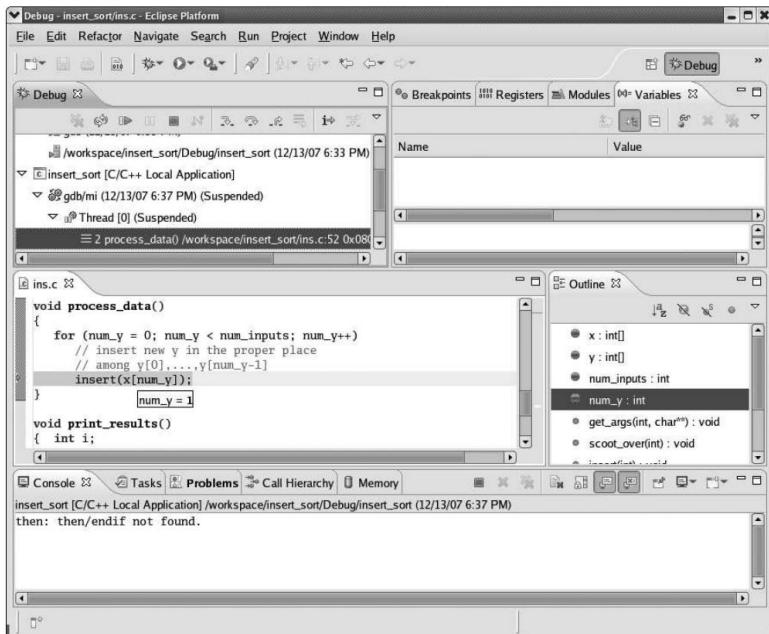


Рисунок 1-6: Инспекция переменная в Затмение

Как вы увидите в Главе 2, в GDB или DDD вы также можете организовать непрерывно отображать переменную, чтобы вам не приходилось повторно запрашивать см. значение. DDD имеет особенно приятную функцию для отображения связанных списков, деревья и другие структуры данных, содержащие указатели: вы можете щелкнуть исходящую ссылку любого узла в такой структуре, чтобы найти следующий узел.

1.5.3 Выдача а "Все Очки Бюллетень" для Изменения К Переменная

Точка наблюдения объединяет в себе понятия точки останова и проверки переменных.

Самая простая форма дает отладчику указание приостанавливать выполнение программы всякий раз, когда изменяется значение указанной переменной.

Например, предположим, что вы хотите проверить состояние программы в точках ее выполнения, в которых изменяется переменная z. значение. В GDB вы можете выполнить команду

(gdb) смотреть z

При запуске программы GDB приостанавливает выполнение всякий раз, когда значение z изменяется. В DDD вы бы установили точку наблюдения, щелкнув любую экземпляр z в окне исходного текста, а затем щелкните значок Watch в верхней части окна DDD.

Еще лучше то, что вы можете устанавливать точки наблюдения на основе условных выражений. Скажем, например, что вы хотите найти первую точку выполнения программы, в которой значение z превышает 28. Вы можете сделать это, установив точку наблюдения на основе выражения ($z > 28$). В GDB вы должны ввести

(gdb) смотреть ($z > 28$)

В DDD вы бы ввели эту команду в Консоль DDD. Помните, что в языке C выражение ($z > 28$) имеет логический тип и может быть оценено как истинное или ложь, где ложь представлена 0, астина представлена любым ненулевым числом целое число, обычно 1. Когда z впервые принимает значение больше 28, значение выражение ($z > 28$) изменится с 0 на 1, и GDB приостановит выполнение программы.

Вы можете установить точку наблюдения в Eclipse, щелкнув правой кнопкой мыши в исходном окне, выбрав «Добавить выражение наблюдения», а затем заполнив нужное выражение в диалоговом окне.

Точки наблюдения обычно не так полезны для локальных переменных, как для переменные с более широкой областью действия, поскольку точка наблюдения, установленная на локальной переменной, отмениается, как только переменная выходит из области видимости, то есть когда функция, в которой определена переменная, завершается. Однако локальные переменные в main() — очевидное исключение, поскольку такие переменные не освобождаются до тех пор, пока программа завершает выполнение.

1.5.4 Движущийся Вверх И Вниз по Вызов Кучи

Во время вызова функции сохраняется информация времени выполнения, связанная с вызовом. в области памяти, известной как стековый кадр. Кадр содержит значения

локальных переменных функции и ее параметров, а также запись о месте, из которого была вызвана функция. Каждый раз, когда происходит вызов функции, создается новый кадр и помещается в стек, поддерживаемый системой; кадр в верхней части стека представляет текущую выполняемую функцию, и он извлекается из стека и освобождается при выходе из функции.

Например, предположим, что вы приостанавливаете выполнение вашего примера программы `insert_sort`, находясь в функции `insert()`. Данные в текущем фрейм стека сообщит, что вы попали туда через вызов функции в определенном месте, которое оказывается внутри функции `process_data()` (которая вызывает `В` фрейме также будет храниться текущее значение единственного локального значения `insert ()`. переменная, которую вы увидите позже, — `j`.

Кадры стека для других активных вызовов функций будут содержать подобная информация, и вы также можете изучить их, если хотите. Например, даже если выполнение в настоящее время находится в `insert()`, вы можете захотеть взглянуть на предыдущий кадр в стеке вызовов, то есть на `process_data()` фрейм. Это можно сделать в GDB с помощью команды

(gdb) кадр 1

При выдаче команды GDB frame кадр текущей выполняемой функции имеет номер 0, ее родительский кадр (то есть кадр стека вызывающей функции) имеет номер 1, родительский элемент родителя имеет номер 2 и т. д. Команда GDB up переносит вас к следующему родительскому элементу в стеке вызовов (например, к кадру 1 из кадра 0), а down переносит вас в другом направлении. Такие операции очень полезны, поскольку значения локальных переменных в Некоторые из более ранних кадров стека могут дать вам подсказку о том, что вызвало ошибку.

Обход стека вызовов не изменяет путь выполнения — в этом случае например, следующая строка `insert_sort`, которая будет выполнена, по-прежнему будет текущей один в `insert()` — но он позволяет вам взглянуть на фреймы-предки и таким образом проверить значения локальных переменных для вызовов функций ведущая к текущей. Опять же, это может дать вам подсказки о том, где найдите ошибку.

Команда GDB backtrace покажет вам весь стек, то есть еп- коллекция шин рам, существующих в настоящее время.

Аналогичная операция в DDD вызывается нажатием кнопки «Состояние» | «Обратная трассировка»; появится окно, отображающее все кадры, и вы можете нажать кнопку какой бы вы ни хотели проверить. Интерфейс DDD также имеет Up и Кнопки «Вниз», которые можно нажимать для вызова команд GDB «вверх» и «вниз».

В Eclipse стек постоянно виден в перспективе Debug. На рисунке 1-7 посмотрите на вкладку Debug в верхнем левом углу. Вы увидите что мы сейчас находимся в кадре 2, в функции `get_args()`, которую мы вызвали из кадра 1 в `main()`. Какой бы кадр ни был выделен, он будет отображаться в исходном окне, так что вы можете отобразить любой кадр, щелкнув по нему в вызове куча.

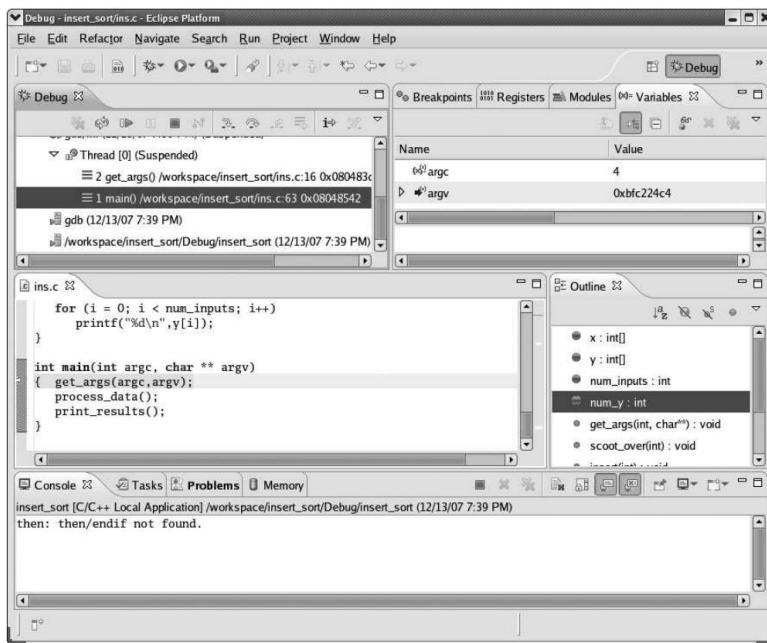


Рисунок 1-7: Перемещение внутри стека в Затмение

1.6 Онлайн-помощь

В GDB доступ к документации можно получить с помощью команды `help`.
Пример,

(gdb) помощь точки останова

даст вам документацию по точкам останова. Команда GDB `help`,
без аргументов, дает вам меню категорий команд, которые могут быть

используются в качестве аргументов для оказания помощи.

В DDD и Eclipse множество материалов доступно при нажатии кнопки «Справка».

1.7 Вводный сеанс отладки

Теперь мы представим полную сессию отладки. Как уже упоминалось, пример программы находится в исходном файле `ins.c` и выполняет сортировку вставкой. Это не эффективный метод сортировки, конечно, но простота кода делает это хорошо для иллюстрации операций отладки. Вот код:

```
//  
// сортировка вставкой, несколько ошибок  
//  
// использование: insert_sort num1 num2 num3 // быть ... , где numi — это числа, которые нужно  
// отсортированным
```

```

int x[10], // входной массив
y[10], // массив рабочей
области num_inputs, // длина входного
массива num_y = 0; // текущее количество элементов в у

void get_args(int ac, char **av) { int i;

    num_inputs = ac - 1; для
    (i = 0; i < num_inputs; i++) x[i] =
        atoi(av[i+1]);
}

void scoot_over(int jj) { int k;

    for (k = num_y-1; k > jj; k++) y[k] =
        y[k-1];
}

void insert(int new_y) { int
j;

    if (num_y = 0) { // у пока пуст, простой случай y[0] =
        new_y; return;

    } // необходимо вставить непосредственно
перед первым элементом у, // которого
new_y меньше for (j = 0; j < num_y;
    j++) { if (new_y < y[j]) { //
        сдвигаем y[j], y[j+1],... вправо // перед
        вставкой new_y scoot_over(j);
        y[j] = new_y;
        return;

    }
}
}

void process_data() {

    for (num_y = 0; num_y < num_inputs; num_y++) //
        вставить новый у в нужное место //
        среди y[0],...,y[num_y-1]
        insert(x[num_y]);
}

```

```

}

void print_results() { int i;

    для (i = 0; i < num_inputs; i++)
        printf("%d\n",y[i]);
}

int main(int argc, char ** argv)
{ get_args(argc,argv);
    process_data();
    print_results();
}

```

Ниже приведено описание псевдокода программы. Вызовы функций обозначены операторами вызова , а псевдокод для каждой функции показан с отступом под вызовами:

вызывать

main(): сделать массив
у пустым вызвать

get_args(): получить num_inputs чисел x[i] из
командной строки

вызвать process_data(): для
i = 1 до num_inputs
вызвать

insert(x[i]): new_y = x[i] найти первый y[j], для
которого new_y < y[j]
вызвать scoot_over(j): сдвинуть y[j],
y[j+1], ... вправо, чтобы
освободить место для new_y установить y[j] = new_y

Давайте скомпилируем и запустим код:

\$ gcc -g -Wall -o вставка_сортировки ins.c

Важно: Вы можете использовать опцию -g для GCC, чтобы указать компилятору сохранить таблицу символов , то есть список адресов памяти, соответствующих переменным и строкам кода вашей программы, в сгенерированном исполняемом файле, который здесь — insert_sort. Это абсолютно необходимый шаг, который позволяет вам ссылаться на имена переменных и номера строк в исходном коде во время сеанса отладки. Без этого шага (и что-то похожее пришлось бы сделать, если бы вы использовали компилятор, отличный от GCC), вы не могли бы попросить отладчик «остановиться на строке 30» или «вывести значение x», например.

Теперь запустим программу. Следуя принципу «Начать с малого» из раздела 1.3.3, сначала попробуйте отсортировать список, состоящий всего из двух чисел:

```
$ вставка_сортировки 12 5
(выполнение остановлено нажатием пользователем ctrl-C)
```

Программа не завершилась и не вывела никаких результатов. По-видимому, она пошла в бесконечный цикл, и нам пришлось его прервать, нажав CTRL-C. Нет сомнения: что-то не так.

В следующих разделах мы сначала представим сеанс отладки для эта глючная программа с использованием GDB, а затем обсудим, как те же операции выполняются с использованием DDD и Eclipse.

1.7.1 The ГБД Подход

Чтобы отследить первую ошибку, запустите программу в GDB и дайте ей поработать некоторое время, прежде чем приостановить его с помощью CTRL-C. Затем посмотрите, где вы находитесь. В этом таким образом можно определить местоположение бесконечного цикла.

Сначала запустите отладчик GDB на insert_sort:

```
$ gdb insert_sort -tui
```

Теперь ваш экран будет выглядеть так:

```
63     { get_args(argc,argv);
64         process_data();
65         print_results();
66     }
67
68
69
Файл: ins.c      Процедура: ??      Линия: ??      ПК: ??
```

(гдб)

Верхнее подокно отображает часть вашего исходного кода, а нижнее в подокне вы видите приглашение GDB, готовое к вашим командам. Есть а также приветственное сообщение GDB, которое мы опустили ради краткости.

Если вы не запросите режим TUI при вызове GDB, вы получите только приветственное сообщение и приглашение GDB, без верхнего подокна для исходного кода вашей программы. Затем вы можете войти в режим TUI, используя команда GDB CTRL-XA. Эта команда переключает вас в TUI и из него режим и полезен, если вы хотите, например, временно выйти из режима TUI чтобы вам было удобнее читать онлайн-справку GDB, или чтобы вы можно просмотреть большую часть истории команд GDB на одном экране.

Теперь запустите программу из GDB, введя команду run вместе с аргументами командной строки вашей программы, а затем нажмите CTRL-C. чтобы приостановить его. Теперь экран выглядит так:

```

46
47 void process_data() 48 { 49
for
( num_y = 0; num_y < num_inputs; num_y++ ) 50 // вставить новый
y в нужное место 51 // среди y[0],...,y[num_y-1] > 52
insert(x[num_y]); 53 }

54
55 void print_results() 56 { int i;

57
58 для (i = 0; i < num_inputs; i++) 59
printf("%d\n",y[i]); 60 } .

```

Файл: ins.c Процедура: process_data Стока: 52 ПК: 0x8048483

(gdb) запустить 12.5
Запуск программы: /debug/insert_sort 12.5

Программа получила сигнал SIGINT, прерывание.
0x08048483 в process_data () в ins.c:52 (gdb)

Это говорит вам, что когда вы остановили программу, insert_sort была в функция process_data() и строка 52 в исходном файле ins.c должны были быть выполнены.

Мы нажали CTRL-C в случайное время и остановились в случайному месте кода. Иногда полезно приостановить и перезапустить программу, которая перестала отвечать два или три раза, выполнив continue между CTRL-C, чтобы увидеть, где вы останавливаетесь каждый раз.

Теперь строка 52 является частью цикла, который начинается в строке 49. Этот цикл бесконечный? Цикл не выглядит так, как будто он должен работать бесконечно, но Принцип Подтверждения говорит, что вы должны проверить это, а не просто предположить. Если цикл не завершается из-за того, что вы каким-то образом неправильно установили верхнюю границу для переменной num_y , то после того, как программа проработает некоторое время, значение num_y будет огромным. Так ли это? (Опять же, похоже, что не должно быть, но вам нужно это подтвердить.) Давайте проверим, каково текущее значение num_y , попросив GDB вывести его.

(gdb) распечатать
num_y \$1 = 1

Вывод этого запроса в GDB показывает, что значение num_y равно 1. Метка \$1 означает, что это первое значение, которое вы попросили GDB вывести. (Значения, обозначенные как \$1, \$2, \$3 и т. д., вместе называются историей значений сеанса отладки. Они могут быть очень полезны, как вы увидите далее)

(главы.) Так что, похоже, мы находимся только на второй итерации цикла строка 49. Если бы этот цикл был бесконечным, он бы уже давно прошел вторую итерацию.

Давайте подробнее рассмотрим, что происходит, когда num_y равен 1. Сообщите GDB остановитесь в insert() во время второй итерации цикла на строке 49, так что вы можете осмотреться и попытаться выяснить, что не так место и время в программе:

(gdb) перерыв 30

Точка останова 1 по адресу 0x80483fc: файл ins.c, строка 30.

(gdb) условие 1 num_y==1

Первая команда устанавливает точку останова на строке 30, то есть в начале insert(). В качестве альтернативы вы могли бы указать эту точку останова через команду break insert, означающая прерывание на первой строке insert() (здесь это строка 30). Эта последняя форма имеет преимущество: если вы измените программный код так, чтобы функция insert() больше не начиналась со строки 30 ins.c , ваша точка останова останется действительной, если указана с помощью функции имени, но не если указано с использованием номера строки.

Обычно команда break останавливает выполнение каждый раз, когда программа достигает указанной строки. Однако вторая команда здесь, condition 1 num_y==1 , делает эту точку останова условной: GDB приостановит выполнение программы в точке останова 1 только тогда, когда выполняется условие num_y==1 .

Обратите внимание, что в отличие от команды break , которая принимает номера строк (или имена функций), условие принимает номер точки останова. Вы всегда можете использовать команду info break для поиска номера нужной точки останова. (Эта команда также дает вам другую полезную информацию, например, номер (количество раз, когда каждая точка останова была достигнута до сих пор.)

Мы могли бы объединить команды break и condition в синтаксис...

gle шаг с использованием break if следующим образом:

(gdb) перерыв 30 если num_y==1

Затем запустите программу снова, используя команду run . У вас нет чтобы переформулировать аргументы командной строки, если вы просто хотите повторно использовать старые. Это как раз тот случай, и поэтому вы можете просто ввести гип. Поскольку программа уже запущен, GDB спрашивает нас, хотите ли вы перезапустить с самого начала, и Вы отвечаете «да».

Теперь экран будет выглядеть так:

```

24          y[k] = y[k-1];
25      }
26
27      недействительная вставка (int new_y)
28      { int j;
29
*> 30      если (num_y == 0) { // у пока пуст, простой случай
31          y[0] = новый_y;

```

```

32             возвращаться;
33         }
34         // нужно вставить непосредственно перед первым у
35         // элемент, new_y которого меньше
36         для (j = 0; j < num_y; j++) {
37             если (новый_y < y[j]) {
38                 // сдвиг y[j], y[j+1]... вправо
Файл: ins.c      Процедура: вставить      Стока: 30      ПК: 0x80483fc
-----
```

```

(gdb) условие 1 num_y==1
(gdb) запустить
Отлаживаемая программа уже запущена.
Начать с самого начала? (да или нет)
Запуск программы: /debug/insert_sort 12 5
```

Точка останова 1, вставить (new_y=5) в ins.c:30
 (гдб)

Снова применяем принцип подтверждения: поскольку num_y равен 1, строка 31
 следует пропустить и перейти к выполнению строки 36. Но нам нужно
 Подтверждаем это, поэтому вводим следующую команду для перехода к следующей строке:

```

24             y[k] = y[k-1];
25         }
26
27         недействительная вставка (int new_y)
28         { int j;
29
* 30         если (num_y = 0) { // у пока пуст, простой случай
31             y[0] = новый_y;
32             возвращаться;
33         }
34         // нужно вставить непосредственно перед первым у
35         // элемент, new_y которого меньше
> 36         для (j = 0; j < num_y; j++) {
37             если (новый_y < y[j]) {
38                 // сдвиг y[j], y[j+1]... вправо
Файл: ins.c      Процедура: вставить      Стока: 36      ПК: 0x8048406
-----
```

```

(gdb) запустить
Отлаживаемая программа уже запущена.
Начать с самого начала? (да или нет)
Запуск программы: /debug/insert_sort 12 5
```

Точка останова 1, вставить (new_y=5) в ins.c:30
 (гдб)

Стрелка в верхнем подокне теперь находится на строке 36, так что наши ожидания подтверждаются; мы действительно пропустили строку 31. Теперь давайте продолжим пошаговое выполнение программы, подтверждая предположения о коде по ходу дела. путь. Теперь вы находитесь в начале цикла, поэтому выполните следующую команду еще раз несколько раз и посмотрите, как продвигается цикл, строка за строкой:

```

39          // перед вставкой new_y
40          scoot_over(j);
41          y[j] = новый_y;
42          возвращаться;
43      }
44  }
> 45  }
46
47  недействительные данные_процесса()
48  {
49      для (num_y = 0; num_y < num_inputs; num_y++)
50          // вставьте новый y в нужное место
51          // среди y[0],...,y[num_y-1]
52          вставить(x[num_y]);
53  }

Файл: ins.c    Процедура: вставить    Стока: 45    ПК: 0x804844d

```

Отлаживаемая программа уже запущена.

Начать с самого начала? (да или нет)

Запуск программы: /debug/insert_sort 12 5

Точка останова 1, вставить (new_y=5) в ins.c:30

(gdb) следующий

(gdb) следующий

(гдб)

Посмотрите, где сейчас находится стрелка в верхнем подокне — мы перешли прямо из строки 37 в строку 45! Это довольно неожиданно. Мы не выполнили даже одна итерация цикла. Помните, однако, что сюрпризы хороши, потому что они дают вам подсказки о том, где находятся ошибки.

Единственный способ, которым цикл в строке 36 мог бы не выполнить ни одной итерации вообще, если условие $j < \text{num_y}$ в строке 36 не выполняется, даже когда j равен 0. Тем не менее, вы знаете, что num_y равен 1, потому что вы находитесь в этой функции сейчас после

наложив условие $\text{num_y} == 1$ на точку останова. Или, по крайней мере, вы

думаю, что вы это знаете. Опять же, вы не подтвердили это. Проверьте это сейчас:

(gdb) распечатать num_y

\$2 = 0

Конечно же, условие $\text{num_y} == 1$ соблюдалось, когда вы ввели `insert()`, но, видимо, num_y с тех пор изменился. Каким-то образом num_y стал равен 0 после вы вошли в эту функцию. Но как?

Как упоминалось ранее, принцип подтверждения не говорит вам, что это за ошибка, но это дает нам подсказки о том, где ошибка, скорее всего, находится. В этом случае вы теперь обнаружили, что местоположение находится где-то между строки 30 и 36. И вы можете сузить этот диапазон еще больше, потому что вы увидел, что строки с 31 по 33 пропущены, а строки с 34 по 35 являются комментариями. Другими словами, загадочное изменение значения в `num_y` произошло либо в строке 30, либо в строке 36.

Сделав небольшой перерыв — часто это лучшая стратегия отладки! — мы внезапно понимаем, что неисправность — это классическая ошибка, часто допускаемая при начале работы (и, как ни странно, опытные программисты на языке C отмечают: в строке 30 мы использовали `=` вместо `==`, превратив проверку на равенство в присваивание).

Видите, как возникает бесконечный цикл? Ошибка в строке 30 устанавливает вечная ситуация качелей, в которой часть `num_y++` строки 49 многократно увеличивает `num_y` от 0 до 1, в то время как ошибка в строке 30 многократно устанавливает значение этой переменной обратно на 0.

Итак, мы исправляем эту унизительную ошибку (а какие из них не унизительны?), перекомпилируем и попробуем запустить программу снова:

```
$ вставка_сортировки 12 5
5
0
```

У нас больше нет бесконечного цикла, но у нас нет правильного выхода либо.

Вспомните из псевдокода, что должна делать ваша программа. Здесь: Изначально массив у пуст. Первая итерация цикла в строке 49 предполагается поместить 12 в `y[0]`. Затем во второй итерации 12 предполагается сместить на одну позицию массива, чтобы освободить место для вставки 5. Вместо этого, похоже, 5 заменило 12.

Проблема возникает со вторым числом (5), поэтому вам следует снова сосредоточиться на второй итерации. Потому что мы мудро решили остаться в GDB сеанс, вместо того, чтобы выйти из GDB после обнаружения и исправления первой ошибки, точка останова и ее условие, которые мы установили ранее, все еще действуют. Таким образом, мы просто запускаем программу снова и останавливаемся, когда программа начинается для обработки второго ввода:

```
24          y[k] = y[k-1];
25      }
26
27      недействительная вставка (int new_y)
28  { int j;
29
*> 30      если (num_y == 0) { // у пока пуст, простой случай
31          y[0] = новый_y;
32          возвращаться;
33      }
34      // нужно вставить непосредственно перед первым у
35      // элемент, new_y которого меньше
```

```

36         для (j = 0; j < num_y; j++) {
37             если (новый_y < y[j]) {
38                 // сдвиг y[j], y[j+1],... вправо Стока: 30
Файл: ins.c      Процедура: вставить                         ПК: 0x80483fc
-----
```

Отлаживаемая программа уже запущена.

Начать с самого начала? (да или нет)

'/debug/insert_sort' изменился; перечитываем символы.

Запуск программы: /debug/insert_sort 12 5

Точка останова 1, вставить (new_y=5) в ins.c:30

(гdb)

Обратите внимание на строку, которая объявляет

'/debug/insert_sort' изменился; перечитываем символы.

Это показывает, что GDB увидел, что мы перекомпилировали программу, и автоматически перезагрузил новый двоичный файл и новую таблицу символов перед запуском программы.

Опять же, тот факт, что нам не пришлось выходить из GDB перед повторной компиляцией нашего Программа очень удобна по нескольким причинам. Во-первых, вам не нужно переформулируйте аргументы командной строки; вы просто вводите run , чтобы повторно запустить программу. Во-вторых, GDB сохраняет точку останова, которую вы установили, так что вы не нужно вводить его снова. Здесь у вас только одна точка останова, но обычно у вас их будет несколько, и тогда это станет настоящей проблемой. Эти удобства избавляют вас от необходимости печатать, и что еще важнее, они избавляют вас от практического отвлекающие факторы и позволяют лучше сосредоточиться на фактической отладке.

Аналогично, вам не следует постоянно выходить из текстового редактора и перезапускать его. во время сеанса отладки, что также будет отвлекать и тратя времени. Просто держите текстовый редактор открытым в одном окне и GDB (или DDD) в другом и используйте третье окно для тестирования вашей программы.

Теперь давайте попробуем пройти код еще раз. Как и прежде, программа следует пропустить строку 31, но, надеюсь, на этот раз он дойдет до строки 37, а не до к ситуации ранее. Давайте проверим это, дважды выполнив следующую команду:

```

31         y[0] = новый_y;
32         возвращаться;
33     }
34     // нужно вставить непосредственно перед первым у
35     // элемент, new_y которого меньше
36     для (j = 0; j < num_y; j++) {
> 37         если (новый_y < y[j]) {
38             // сдвиг y[j], y[j+1],... вправо
39             // перед вставкой new_y
40             scoot_over(j);
41             y[j] = новый_y;
```

```

42             возвращаться;
43         }
44     }
45 }

Файл: ins.c      Процедура: вставить      Стока: 37      ПК: 0x8048423
-----
```

'/debug/insert_sort' изменился; перечитываем символы.

Запуск программы: /debug/insert_sort 12 5

Точка останова 1, вставить (new_y=5) в ins.c:30

(gdb) следующий

(gdb) следующий

(гдб)

Мы действительно достигли 37-й строки.

На данный момент мы считаем, что условие в if в строке 37 должно выполняться, потому что new_y должен быть равен 5, а y[0] должен быть равен 12 с первой итерации.

Вывод GDB подтверждает первое предположение. Давайте проверим второе:

(gdb) распечатать y[0]

3 доллара = 12

Теперь, когда это предположение также подтверждено, выполните следующую команду: что приводит нас к строке 40. Функция scoot_over() должна смешать 12 в следующую позицию массива, чтобы освободить место для 5. Вы должны проверить чтобы увидеть, так ли это. Здесь вы сталкиваетесь с важным выбором. Вы могли бы снова введите следующую команду, что заставит GDB остановиться на строке 41; функция scoot_over() будет выполнена, но GDB не остановится в течение эта функция. Однако, если бы вы вместо этого дали команду step , GDB остановился бы на строке 23, и это позволило бы вам выполнить пошаговое выполнение внутри scoot_over().

Следуя нисходящему подходу к отладке, описанному в разделе 1.3.3, мы выбираем следующую команду вместо шага в строке 40. Когда GDB останавливается на строке 41, вы можете взглянуть на y , чтобы увидеть, выполнила ли функция свое job правильно. Если эта гипотеза подтвердится, вам удастся избежать трудоемкой проверки подробной работы функции scoot_over() это не способствовало бы исправлению текущей ошибки. Если вы не справитесь Чтобы убедиться, что функция работает правильно, вы можете запустить программу в отладчик снова и войдите в функцию, используя step , чтобы проверить подробно рассмотреть работу функции и, надеюсь, определить, где она дает сбой.

Итак, когда вы достигнете строки 40, введите следующую строку, получив

```

31         y[0] = новый_y;
32         возвращаться;
33     }
34     // нужно вставить непосредственно перед первым y
35     // элемент, new_y которого меньше
36     для (j = 0; j < num_y; j++) {
37         если (новый_y < y[j]) {
```

```

38          // сдвиг y[j], y[j+1],... вправо
39          // перед вставкой new_y
40          scoot_over(j);
> 41          y[j] = новый_y;
42          возвращаться;
43      }
44  }
45 }

Файл: ins.c      Процедура: вставить      Стока: 41      ПК: 0x8048440
-----
```

(gdb) следующий

(gdb) следующий

(гдб)

Правильно ли scoot_over() сдвинул 12? Давайте проверим:

```
(gdb) распечатать у
$4 = {12, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Видимо нет. Проблема действительно в scoot_over(). Давайте удалим точку останова в начале insert() и поместите ее в scoot_over(), снова с условием, что мы остановимся там во время второй итерации строки 49:

```
(gdb) очистить 30
Удален остановочный пункт 1
(gdb) перерыв 23
Точка останова 2 по адресу 0x80483c3: файл ins.c, строка 23.
(gdb) условие 2 num_y==1
```

Теперь запустите программу еще раз:

```

15      num_inputs = ac - 1;
16      для (i = 0; i < num_inputs; i++)
17          x[i] = atoi(av[i+1]);
18      }
19
20      void scoot_over(int jj)
21      { int k;
22
*> 23          для (k = num_y-1; k > jj; k++)
24              y[k] = y[k-1];
25          }
26
27          недействительная вставка (int new_y)
28      { int j;
29

Файл: ins.c      Процедура: scoot_over      Стока: 23      ПК: 0x80483c3
```

```
(gdb) условие 2 num_y==1
(gdb) запустить
Отлаживаемая программа уже запущена.
Начать с самого начала? (да или нет)
Запуск программы: /debug/insert_sort 12 5
```

Точка останова 2, scoot_over (jj=0) в ins.c:23
(гдб)

Еще раз следуйте принципу подтверждения: подумайте о том, что вы ожидаете, что это произойдет, а затем попытайтесь подтвердить, что это действительно происходит. В этом случае, Предполагается, что функция сдвигает 12 на следующую позицию в массиве y, что означает, что цикл в строке 23 должен пройти ровно через один итерация. Давайте пройдемся по программе, многократно выдавая следующую команду, чтобы проверить это ожидание:

```
15      num_inputs = ac - 1;
16      для (i = 0; i < num_inputs; i++)
17          x[i] = atoi(av[i+1]);
18      }
19
20      void scoot_over(int jj)
21      { int k;
22
*23      для (k = num_y-1; k > jj; k++)
24          y[k] = y[k-1];
>25      }
26
27      недействительная вставка (int new_y)
28      { int j;
29
Файл: ins.c      Процедура: scoot_over      Стока: 25      ПК: 0x80483f1
```

Отлаживаемая программа уже запущена.
Начать с самого начала? (да или нет)
Запуск программы: /debug/insert_sort 12 5

Точка останова 2, scoot_over (jj=0) в ins.c:23
(gdb) следующий
(gdb) следующий
(гдб)

Здесь нас снова ждет сюрприз: теперь мы находимся на строке 25, ни разу не коснувшись строки 24 — цикл не выполнил ни одной итерации, ни одной итерации, которую мы ожидали, что он выполнится. Видимо, в строке 23 есть ошибка.

Как и в предыдущем цикле, который неожиданно не выполнил ни одной итерации body, должно быть, условие цикла не было выполнено в самом начале цикла. Это так? Условие цикла в строке 23 — k >

jj. Мы также знаем из этой строки, что начальное значение k равно num_y-1, и мы знаем, из нашего условия точки останова следует, что последняя величина равна 0. Наконец, GDB экран сообщает нам, что jj равен 0. Таким образом, условие k > jj не было выполнено, когда цикл начался.

Таким образом, мы неправильно указали либо условие цикла k > jj , либо инициализацию k = num_y-1. Учитывая, что 12 должно было переместиться из y[0] к y[1] в первой и единственной итерации цикла, то есть строка 24 должна выполнили с k=1 — мы понимаем, что инициализация цикла неверна. Должно было быть k = num_y.

Исправьте ошибку, перекомпилируйте программу и запустите ее снова (вне GDB):

```
$ вставка_сортировки 12 5
Ошибка сегментации
```

Ошибки сегментации, подробно обсуждаемые в главе 4, возникают, когда запущенная программа пытается получить доступ к памяти, на которую у нее нет разрешения. для доступа. Обычно причиной является выход индекса массива за пределы допустимого диапазона или ошибочный значение указателя. Ошибки сегмента могут также возникать из-за ссылок на память, которые не явно включают указатель или переменные массива. Одним из примеров этого может быть еще одна классическая ошибка программиста на языке C, забывшего амперсанд в параметре функции, который передается с помощью вызова по ссылке, например, письмо

```
scanf("%d",x);
```

вместо

```
scanf("%d",&x);
```

В общем, основная ценность отладочного инструмента, такого как GDB или DDD, заключается в том, чтобы облегчить процесс проверки предположений кодирования, но в случае сбоев сегмента отладочный инструмент дает дополнительную, ощутимую, немедленную помощь: он сообщает, где в вашей программе произошла ошибка.

Чтобы воспользоваться этим, вам нужно запустить insert_sort в GDB и воссоздать ошибку seg. Сначала удалите точку останова. Как было показано ранее, чтобы сделать это вам нужно указать номер строки точки останова. Вы, возможно, уже помните это, но найти ее легко: либо прокрутите окно TUI (используя клавиши со стрелками вверх и вниз), ища строки, отмеченные звездочками, либо используйте команду GDB info break . Затем удалите точку останова, используя команду очистки :

```
(gdb) очистить 30
```

Теперь запустите программу снова в GDB:

```

19
20     void scoot_over(int jj)
21     { int k;
22
23         для (k = num_y; k > jj; k++)
24             y[k] = y[k-1];
25     }
26
27     недействительная вставка (int new_y)
28     { int j;
29
30         если (num_y == 0) { // y пока пуст, простой случай
31             y[0] = новый_y;

```

Файл: ins.c Процедура: scoot_over Линия: 24 ПК: 0x08048538

Начать с самого начала? (да или нет)

'/debug/insert_sort' изменился; перечитываем символы.
Запуск программы: /debug/insert_sort 12 5

Программа получила сигнал SIGSEGV, ошибка сегментации.
0x08048538 в scoot_over (jj=0) в ins.c:24
(гдб)

Как и было обещано, GDB сообщает нам, где именно произошла ошибка сегмента, в строке 24, и конечно же, индекс массива, по-видимому, задействован, а именно k. Либо k было достаточно большим, чтобы превысить число элементов в y, или k-1 было отрицательным. Очевидно, что первым делом нужно определить значение k:

(gdb) печать k
4 доллара = 584

Ого! Код задал размерность у так, чтобы она содержала всего 10 элементов, так что это значение k действительно далеко за пределами диапазона. Теперь нам нужно отследить причину. Прежде всего, определите итерацию большого цикла в строке 49 во время в котором произошел этот сбой сегмента.

(gdb) распечатать num_y
5 долларов = 1

Так было во время второй итерации, которая является первым разом, когда выполняется функция scoot_over(). Другими словами, это не тот случай, когда строка 23 работал нормально в первых нескольких вызовах scoot_over(), но позже перестал работать. Есть все еще что-то фундаментально неправильно в этой строке кода. И поскольку единственным оставшимся кандидатом является утверждение

К++

(вспомните, что вы проверили две другие части этой строки ранее), она должна быть виновником. Сделав еще один перерыв, чтобы прочистить голову, мы понимаем, некое смущение, что это должно было быть `k--`.

Исправьте эту строку, еще раз перекомпилируйте и запустите программу:

```
$ вставка_сортировки 12 5
5
12
```

Вот это прогресс! Но работает ли программа для большего набора данных?

Давайте попробуем один вариант:

```
$ вставка_сортировки 12 5 19 22 6 1
1
5
6
12
0
0
```

Теперь вы можете начать видеть свет в конце туннеля. Большинство массив сортируется правильно. Первое число в списке, которое не попадает отсортировано правильно — 19, поэтому установите точку останова на строке 36, на этот раз с условием `new_y == 19`:

```
(gdb) б 36
Точка останова 3 по адресу 0x804840d: файл ins.c, строка 36.
(gdb) условие 3 new_y==19
```

Затем запустите программу в GDB (убедитесь, что вы используете те же аргументы, 12 5 19 22 6 1). Когда вы достигнете точки останова, вы подтвердите, что массив `y` был отсортирован правильно до этого момента:

```
31         y[0] = новый_y;
32         возвращаться;
33     }
34     // нужно вставить непосредственно перед первым y
35     // элемент, new_y которого меньше
*> 36     для (j = 0; j < num_y; j++) {
37         если (новый_y < y[j]) {
38             // сдвиг y[j], y[j+1],... вправо
39             // перед вставкой new_y
40             scoot_over(j);
41             y[j] = новый_y;
```

¹ Пришло время начать использовать общепринятые сокращения для команд. К ним относятся `b` для перерыва, `lb` для информационного перерыва, `cond` для условия, `p` для запуска, `n` для следующего, `s` для шага, `c` для продолжения, `r` для `print` и `bt` для обратной трассировки.

```

42
43      }
Файл: ins.c    Процедура: вставить    Стока: 36    ПК: 0x8048564
-----
```

Начать с самого начала? (да или нет)

Запуск программы: /debug/insert_sort 12 5 19 22 6 1

Точка останова 2, вставить (new_y=19) в ins.c:36

```

(gdb) py
$1 = {5, 12, 0, 0, 0, 0, 0, 0, 0, 0}
(гдб)
```

Пока все хорошо. Теперь попробуем определить, как программа глотает вверх по 19. Мы будем проходить код по одной строке за раз. Обратите внимание, что поскольку 19 не меньше 5 или 12, мы не ожидаем, что условие в if оператор в строке 37 для удержания. После нажатия n несколько раз, мы оказываемся на строка 45:

```

35      // элемент, new_y которого меньше
* 36      для (j = 0; j < num_y; j++) {
37          если (новый_y < y[j]) {
38              // сдвиг y[j], y[j+1],... вправо
39              // перед вставкой new_y
40              scoot_over(j);
41              y[j] = новый_y;
42          возвращаться;
43      }
44  }
> 45  }
46
47  недействительные данные_процесса)
Файл: ins.c Процедура: вставить    Стока: 45    ПК: 0x80485c4
-----
```

```

(гдб) н
(гдб) н
(гдб) н
(гдб) н
(гдб) н
(гдб)
```

Мы на строке 45, собираемся выйти из цикла, не сделав ничего с 19 вообще! Некоторые проверки показывают, что наш код не был написан чтобы охватить важный случай, а именно случай, когда new_y больше любого элемента, который мы обработали до сих пор — упоминание, также выявленное в комментариях к строки 34 и 35:

```
// нужно вставить непосредственно перед первым у
// элемент, new_y которого меньше
```

Чтобы обработать этот случай, добавьте следующий код сразу после строки 44:

```
// еще один случай: new_y > все существующие элементы у
y[num_y] = новый_y;
```

Затем перекомпилируйте и попробуйте еще раз:

```
$ вставка_сортировки 12 5 19 22 6 1
1
5
6
12
19
22
```

Это правильный вывод, и последующее тестирование дает правильные результаты. хорошо.

1.7.2 The Такой же Сессия в DDD

Давайте посмотрим, как приведенный выше сеанс GDB был бы реализован в DDD.

Конечно, нет необходимости повторять все шаги; просто сосредоточьтесь на отличиях от GDB.

Запуск DDD аналогичен запуску GDB. Скомпилируйте исходный код с помощью GCC с опцией -g , а затем введите

```
$ ддд вставка_сортировка
```

для вызова DDD. В GDB вы начали выполнение программы через run команда, включая аргументы, если таковые имеются. В DDD вы нажимаете Программа | Выполнить, после чего вы увидите экран, показанный на рисунке 1-8.

Появилось окно «Выполнить», в котором представлена история предыдущих наборов аргументов командной строки, которые вы использовали. Пока нет предыдущих наборов, но если бы они были, вы могли бы выбрать один из них, щелкнув по нему, или вы могли бы ввести новый набор аргументов, как показано здесь. Затем нажмите «Выполнить».

В сеансе отладки GDB мы некоторое время запускали нашу программу в отладчик, а затем приостановил его с помощью CTRL-C, чтобы исследовать по-видимому, бесконечный цикл. В DDD мы приостанавливаем программу, нажимая Inter-rupt в Command Tool. Экран DDD теперь выглядит так, как на Рис. 1-9. Поскольку DDD действует как интерфейс для GDB, этот щелчок мыши транслируется в операцию CTRL-C в GDB, которую можно увидеть в Консоли.

Следующим шагом в сеансе GDB, описанном выше, была проверка переменной num_y. Как было показано ранее в разделе 1.5, в DDD это делается путем перемещения мыши. указатель на любой экземпляр num_y в исходном окне.

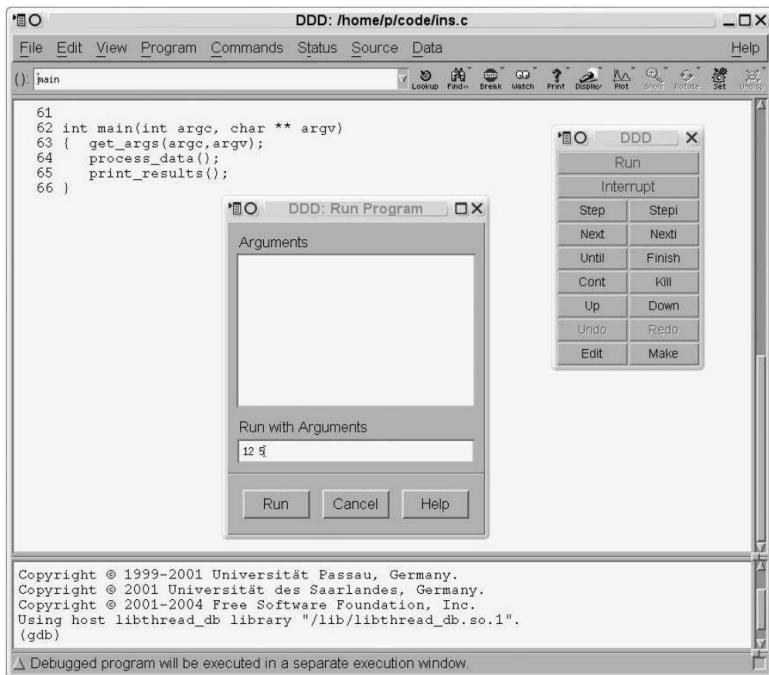


Рисунок 1-8: ДДД Бегать команда

Вы также можете проверять целые массивы таким же образом. Например, в какой-то момент сеанса GDB вы распечатали весь массив `u`. В DDD вы просто переместили бы указатель мыши на любой экземпляр `u` в окне исходного кода. Если вы переместите курсор на `u` в выражении `u[j]` в строке 30, экран будет выглядеть так, как показано на рисунке 1-10. Рядом с этой строкой появилось поле подсказки значения, показывающее содержимое `u`.

Следующим вашим действием в сеансе GDB была установка точки останова на строке 30. Мы уже объяснили, как устанавливать точки останова в DDD, но как насчет установки условия для точки останова, как это было необходимо в этом случае? Вы можете установить условие, щелкнув правой кнопкой мыши значок знака остановки в строке точки останова и выбрав Свойства. Появится всплывающее окно, как показано на рисунке 1-11.

Затем введите ваше условие, `num_u==1`.

Чтобы затем повторно запустить программу, вы должны нажать кнопку Run в Command Tool. Как и в случае с командой GDB `run` без аргументов, эта кнопка запускает программу с последним набором аргументов, который был предоставлен.

Аналогами команд GDB `n` и `s` в DDD являются кнопки Next и Step. тонн в Command Tool. Аналогом `c` в GDB является кнопка Cont.

Этого обзора достаточно, чтобы начать работу с DDD. В последующих главах мы рассмотрим некоторые из расширенных возможностей DDD, такие как его чрезвычайно полезная возможность визуального отображения сложных структур данных, таких как связанные списки и двоичные деревья.

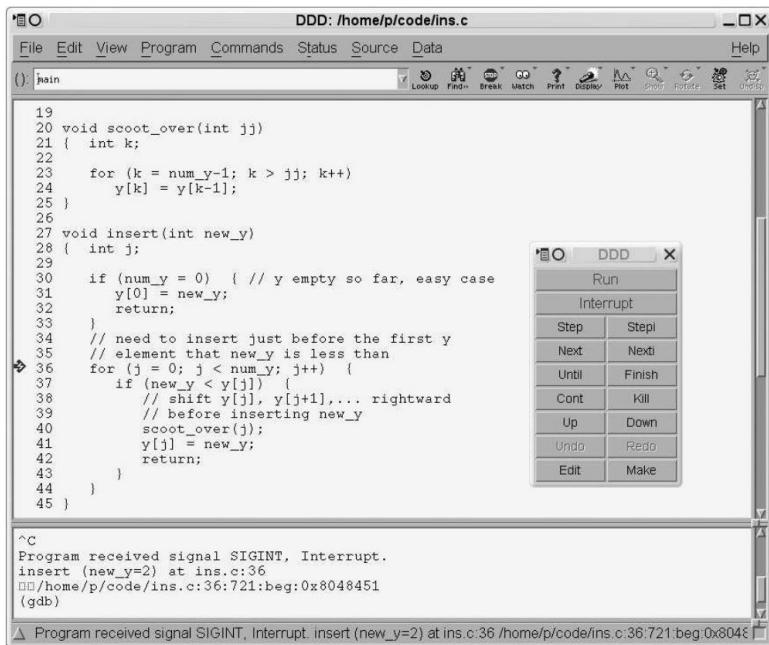


Рисунок 1-9: прерывание

1.7.3 В Затмении

Теперь давайте посмотрим, как бы был реализован приведенный выше сеанс GDB в

Затмение. Как и в нашей презентации по DDD, нет необходимости повторять все шаги; мы просто сосредоточимся на отличиях от GDB.

Обратите внимание, что Eclipse может быть довольно привередливым. Хотя он предлагает много способов выполнить определенную задачу, если не соблюдать строго необходимую последовательность шагов, вы можете оказаться в затруднительном положении, не имея никакого интуитивного решения, кроме для перезапуска части процесса отладки.

Мы предполагаем, что вы уже создали свой проект C/C++.2

При первом запуске/отладке вашей программы вам понадобятся конфигурации запуска и отладки.

Они указывают имя вашего исполняемого файла (и что проект, к которому он принадлежит), его аргументы командной строки (если есть), его специальная оболочка переменная среда (если есть), отладчик по вашему выбору и т. д. Запуск

Конфигурация используется для запуска вашей программы вне отладчика, в то время как

Конфигурация отладки используется в отладчике. Убедитесь, что вы создали оба конфигурации в следующем порядке:

1. Выберите «Выполнить» | «Открыть диалоговое окно «Выполнить»».

2 Поскольку эта книга посвящена отладке, а не управлению проектами, мы не будем здесь много говорить.

о создании и построении проектов в Eclipse. Однако краткое резюме будет таким: вы создайте проект следующим образом: выберите Файл | Новый | Проект; выберите Проект С (или C++); заполните проект имя; выберите Исполняемый файл | Готово. Makefile создается автоматически. Вы собираете (т.е. компилируете) и свяжите свой проект, выбрав Проект | Собрать проект.

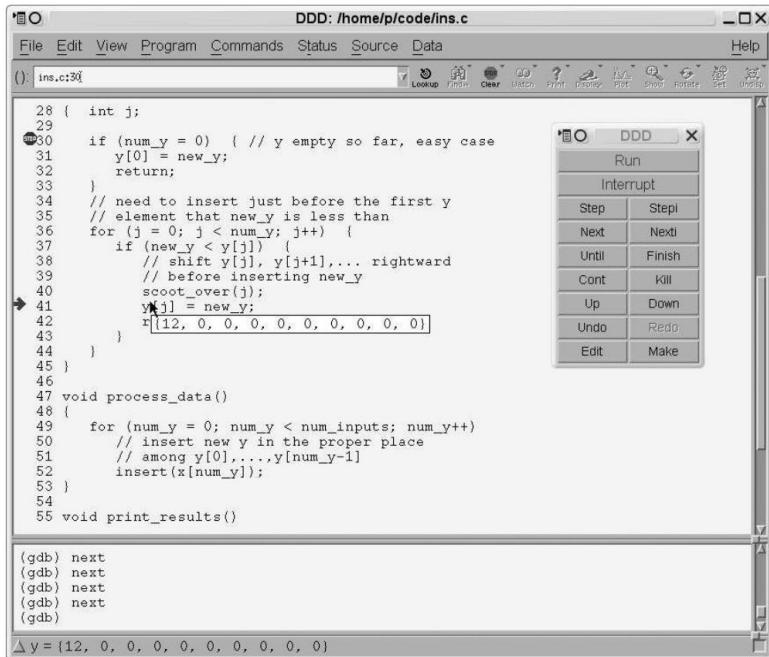


Рисунок 1-10: Проверка массива

2. Щелкните правой кнопкой мыши C/C++ Local Applications и выберите New.
 3. Выберите вкладку «Главная» и заполните конфигурацию запуска, имена проекта и исполняемого файла (Eclipse, вероятно, предложит их вам) и проверьте подключение входа и выхода процесса к клеммной коробке, если вы иметь терминальный ввод/вывод.
 4. Если у вас есть аргументы командной строки или специальные переменные среды, щелкните вкладку «Аргументы» или «Среда» и заполните необходимые параметры.
 5. Выберите вкладку Debugger, чтобы увидеть, какой отладчик используется. Вам, вероятно, не придется это трогать, но хорошо понимать, что есть базовый отладчик, вероятно GDB.
 6. Нажмите «Применить» (если потребуется) и «Закрыть», чтобы завершить создание конфигурации запуска.
 7. Начните создание конфигурации отладки, выбрав «Выполнить» | «Открыть отладку». Диалог Eclipse, вероятно, повторно использует информацию, которую вы предоставили в вашем запустите конфигурацию, как показано на рисунке 1-12, или вы можете изменить ее, если хотите хотите. Снова нажмите «Применить» (если будет предложено) и «Закрыть», чтобы завершить создание вашего отладочная конфигурация.
- Можно создать несколько конфигураций запуска/отладки, обычно с различными наборами аргументов командной строки.

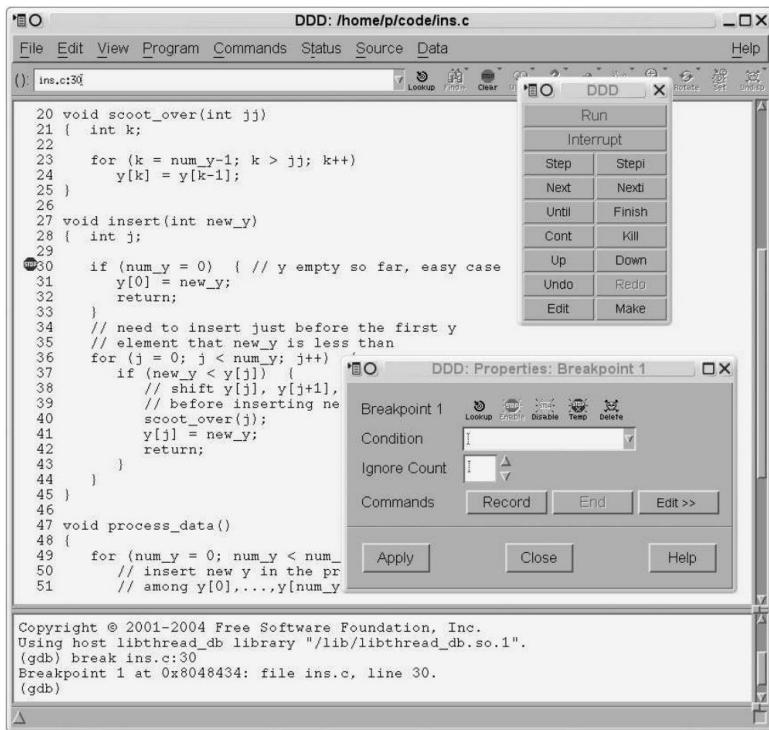


Рисунок 1-11: Внушительный а состояния на точка останова

Чтобы начать сеанс отладки, необходимо перейти в режим отладки, выбрав Окно | Открыть перспективу | Отладка. (Существуют различные (Сокращенные пути, которые мы предоставим вам самим открыть.)

В первый раз, когда вы фактически выполняете действие запуска или отладки, вы делаете это через Выполнить | Открыть диалоговое окно «Выполнить» или Выполнить | Открыть диалоговое окно «Отладка» еще раз, в зависимости от обстоятельств. быть, чтобы указать, какую конфигурацию использовать. После этого, однако, просто выберите Run | Run или Run | Debug, любой из которых перезапустит последнюю отладку конфигурация.

На самом деле, в случае отладки есть более быстрый способ запустить отладочный прогон, Для этого нужно нажать на значок «Отладка» прямо под кнопкой «Навигация» (см. рисунок 1-13). Однако обратите внимание, что всякий раз, когда вы начинаете новый отладочный прогон, вам необходимо чтобы завершить существующие, нажмите на красный квадрат «Завершить», один из них находится на панели инструментов в представлении Debug, а другой — в представлении Console. Представление Debug также имеет значок с двумя крестиками. Удалить все прерванные запуски.

На рисунке 1-13 показан экран, который появится после запуска вашего отладка. Можно задать начальную строку в диалоговых окнах отладки Eclipse, но обычно они по умолчанию устанавливают автоматическую точку останова на первой исполняемой строке кода. На рисунке вы можете увидеть это по символу точки останова в левое поле строки

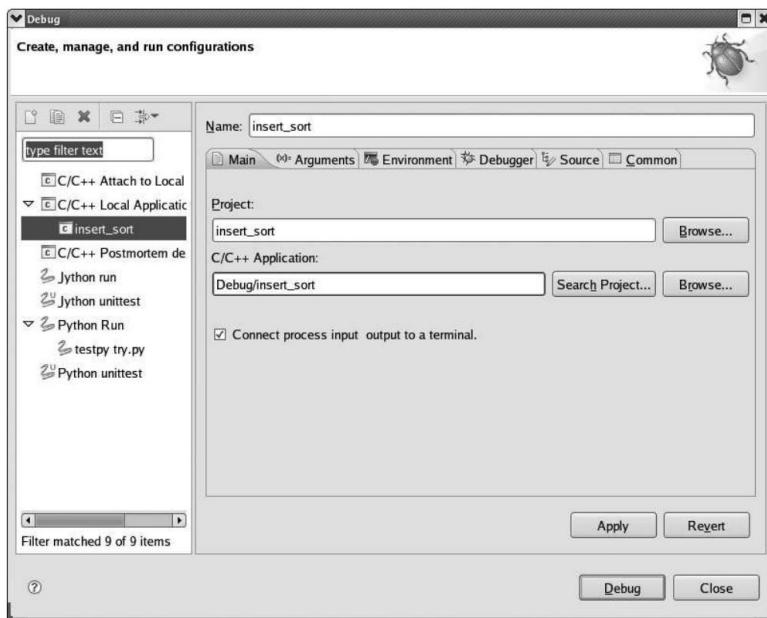


Рисунок 1-12: Диалоговоокно конфигурации отладки

```
{ get_args(argc,argv);
```

Эта строка также подсвечивается, поскольку именно ее вы собираетесь выполнить. Продолжайте и выполните его, нажав на значок «Возобновить» на панели инструментов представления «Отладка» (над полем, которое появилось в окне, когда вы навели указатель мыши на этот значок).

Напомним, что в примере сеанса GDB первая версия программы имел бесконечный цикл, и программа зависла. Здесь, конечно, вы увидите тот же симптом, без вывода в представлении Консоль. Вам нужно завершить программу. Однако вы не хотите делать это, нажимая один из красных квадратов Завершить, потому что это также завершит ваш основной сеанс GDB. Вы хотите остаться в GDB, чтобы посмотреть, где вы были в коде, т. е. расположение бесконечного цикла, проверить значения переменных и т. д. Поэтому вместо операции Завершить выберите Приостановить, нажав значок справа от Возобновить на панели инструментов представления Отладка.

(В литературе по Eclipse эта кнопка иногда называется «Пауза», поскольку ее символ похож на символ для операций паузы в медиаплеерах.)

После нажатия кнопки «Приостановить» ваш экран будет выглядеть так, как показано на рисунке 1-14. Вы увидите что как раз перед этой операцией Eclipse собирался выполнить строку

```
для (j = 0; j < num_y; j++) {
```

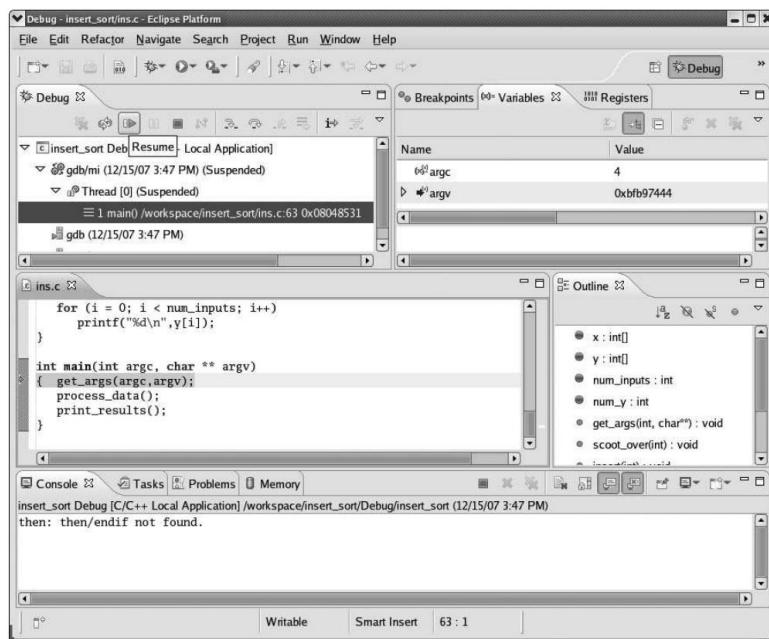


Рисунок 1-13: Начинать отладки

Теперь вы можете проверить значение num_y, переместив указатель мыши на любой экземпляр этой переменной в исходном окне (вы обнаружите, что значение равно 0) и так далее.

Вспомним еще раз нашу сессию GDB выше. После исправления пары ошибок, ваш Программа затем имела ошибку сегментации. Рисунок 1-15 показывает ваш Eclipse экран в этот момент.

Произошло следующее: мы нажали «Возобновить», и наша программа бежал, но внезапно остановился на линии

y[k] = y[k-1];

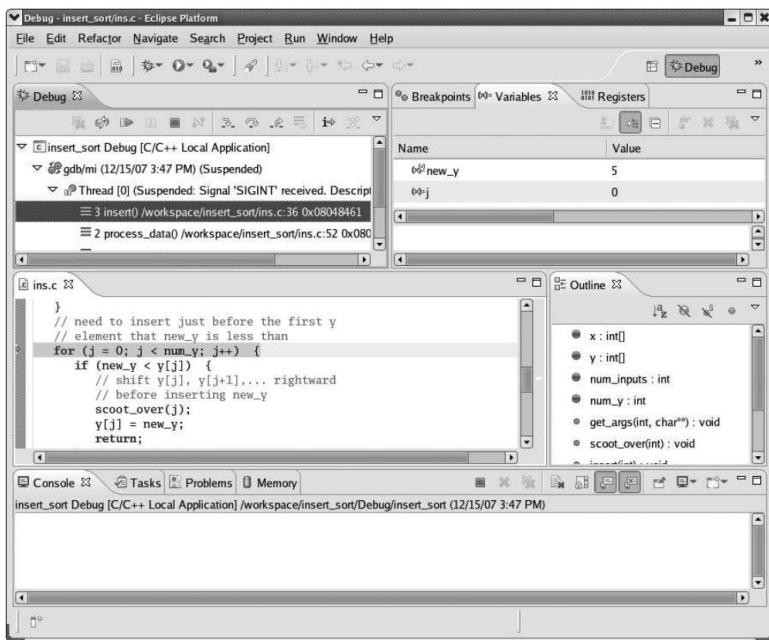
из-за ошибки сегмента. Как ни странно, Eclipse не объявляет об этом в Problems на вкладке «Отладка», но это происходит на вкладке «Отладка» с сообщением об ошибке

(Получен приостановленный сигнал 'SIGSEGV'. Описание: Ошибка сегментации.)

снова видно на рисунке 1-15.

На этой вкладке вы видите, что ошибка произошла в функции scoot_over(), который был вызван из insert(). Снова вы можете запросить значения переменные и обнаружить, например, что k = 544 — далеко за пределами диапазона, как в Пример GDB.

В примере GDB вы также устанавливаете условные точки останова. Вспомните, что в Eclipse вы устанавливаете точку останова, дважды щелкнув в левом поле



Фигура 1-14: прифотографирано

желаемая строка. Чтобы сделать эту точку останова условной, щелкните правой кнопкой мыши символ точки останова для этой строки и выберите Свойства точки останова... | Создать | Common, и заполните условие в диалоге. Диалог изображен на Рисунок 1-16.

Вспомните также, что в вашем сеансе GDB вы иногда запускали свою программу вне GDB, в отдельном окне терминала. Вы можете легко сделать это в Eclipse тоже, выбрав Run | Run. Результаты будут в представлении Console, по-прежнему.

1.8 Использование файлов запуска

Как упоминалось ранее, обычно хорошей идеей будет не выходить из GDB, пока вы перекомпилируете свой код. Таким образом, ваши точки останова и различные другие действия, которые вы настройки (например, команды отображения , которые будут обсуждаться в Главе 3) сохраняются. Если вы вышли из GDB, вам придется вводить их снова.

Однако вам может потребоваться выйти из GDB до завершения отладки. Если вы выходите на перерыв или на день и не можете оставаться в системе на компьютер, вам нужно будет выйти из GDB. Чтобы не потерять их, вы можете поместите команды для точек останова и других настроек в файл запуска GDB, и тогда они будут автоматически загружаться каждый раз при запуске GDB.

Файлы запуска GDB по умолчанию называются .gdbinit . Вы можете иметь один в вашем домашнем каталоге для общих целей и еще один в каталоге содержащий конкретный проект для целей, характерных для этого проекта. Для ин-

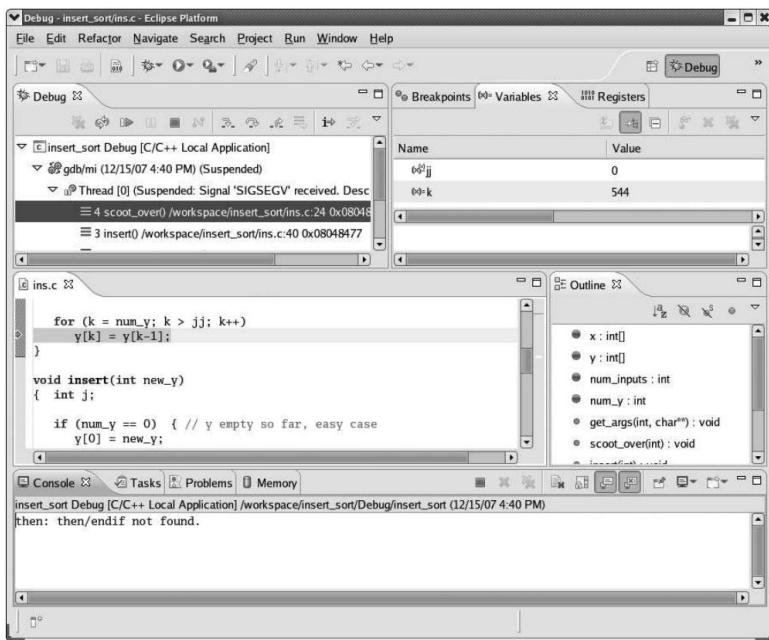


Рисунок1-15: Сегмент вина

позиция, вы бы поместили команды для установки точек останова в файл запуска в последний каталог. В вашем файле .gdbinit в вашем домашнем каталоге вы можете захотеть для хранения некоторых макросов общего назначения, которые вы разработали, как обсуждалось в Глава 2.

GDB считывает файл запуска в вашем домашнем каталоге перед загрузкой исполняемого файла. Так что если бы у вас была команда в .gdbinit вашего домашнего каталога файл, такой как

перерыв г

говоря, что нужно прерваться на функции g(), то GDB всегда будет жаловаться на запуск, что он не знает эту функцию. Однако, линия была бы в порядке в файле запуска локального каталога проекта, поскольку локальный файл запуска — это считывается после загрузки исполняемого файла (и его таблицы символов). Обратите внимание, что эта особенность GDB подразумевает, что лучше не помещать проекты программирования в ваш домашний каталог, так как вы не сможете поместить информацию, специфичную для проекта, в .gdbinit.

Вы можете указать файл запуска во время вызова GDB. Например,

\$ gdb -команда=zx

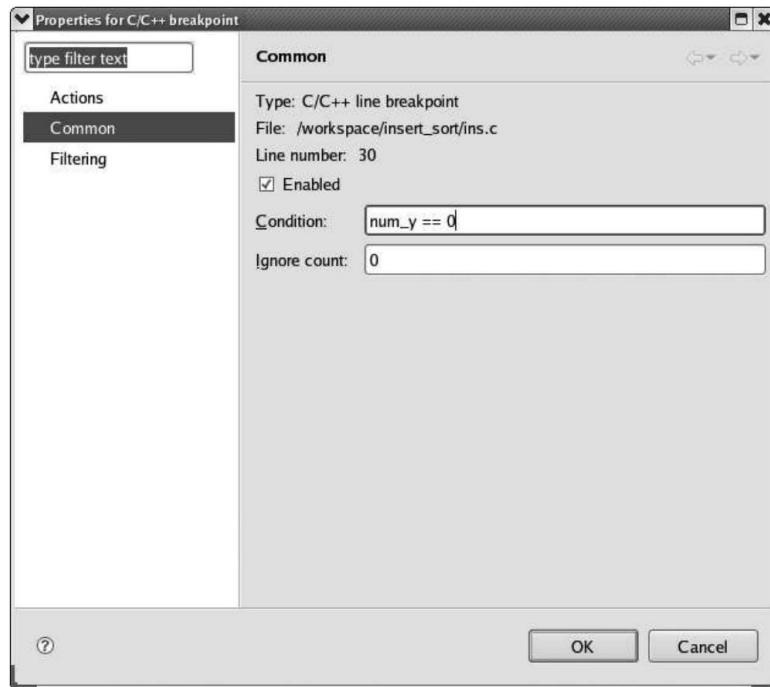


Рисунок1-16: Изготовление ^а условная точка останова

сказал бы запустить GDB на исполняемом файле x, первое чтение в командах из файла z. Кроме того, поскольку DDD — это всего лишь интерфейс для GDB, вызов DDD также вызовет файлы запуска GDB.

Наконец, вы можете настроить DDD различными способами, выбрав Edit | Preferences. Для Eclipse последовательность выглядит так: Window | Preferences.

2

ОСТАНАВЛИВАЮСЬ, ЧТОБЫ ОЗНАКОМИТЬСЯ ВОКРУГ



Символический отладчик, такой как GDB, может запустить вашу программу, как и вы. Однако, с помощью магии включения отладочных символов в исполняемый файл, отладчик создает иллюзию выполнения программы строки за строкой исходного кода, а не инструкция за инструкцией скомпилированного машинного кода. Этот, казалось бы, скромный факт как раз и делает символический отладчик таким полезным при отладке программ.

Если бы все, что мог делать отладчик, это запускать программу, он был бы нам не очень полезен. Мы, конечно, могли бы делать то же самое, и даже более эффективно. Полезность отладчика заключается в том, что мы можем дать ему указание приостановить выполнение программы. После приостановки отладчик дает нам возможность проверить переменные, отследить путь выполнения и многое другое.

2.1 Механизмы паузы

Есть три способа заставить GDB приостановить выполнение вашей программы:

- Точка останова сообщает GDB о необходимости приостановить выполнение в определенном месте внутри программы.
- Точка наблюдения сообщает GDB о необходимости приостановить выполнение, когда определенная ячейка памяти (или выражение, включающее одну или несколько ячеек) изменяет значение.
- Точка перехвата сообщает GDB о необходимости приостановить выполнение при возникновении определенного события.

Поначалу это может сбивать с толку, но все три механизма в документации GDB называются точками останова. Возможно, это связано с тем, что они используют много одинаковых атрибутов и команд.

Например, вы узнаете о команде GDB `delete`, которая, как говорится в аннотации справки, удаляет точку останова:

```
(gdb) help delete Удалить
некоторые точки останова или выражения автоматического отображения.

Аргументы — это номера точек останова с пробелами между ними.

чтобы удалить все точки останова, не указывайте аргумент.
```

Однако опытный пользователь GDB знает, что в справке на самом деле говорится, что команда `Delete` удаляет точки останова, наблюдения и перехвата!

2.2 Обзор точек останова

Точка останова подобна растяжке внутри программы: вы устанавливаете точку останова в определенном «месте» внутри вашей программы, и когда выполнение достигает этой точки, отладчик приостанавливает выполнение программы (и, в случае текстового отладчика, такого как GDB, выдает вам командную строку).

GDB очень гибко трактует значение слова «место»; оно может означать самые разные вещи, например строку исходного кода, адрес кода, номер строки в исходном файле или вход в функцию.

Ниже показан фрагмент сеанса отладки, иллюстрирующий, что происходит, когда GDB останавливается на строке кода. В этом фрагменте мы перечисляем часть исходного кода, ставим точку останова на строке 35 программы, а затем запускаем программу. GDB достигает точки останова и останавливается.

```
(gdb) список
30
31             /* Получить размер файла в байтах */ if ((fd =
32                 open(c.filename, O_RDONLY)) == -1) (void) die(1, "Невозможно
33                 открыть файл."); (void) stat(c.filename, &fstat);
34             cfilesize = fstat.st_size;
35
36
(gdb) break 35 Точка
останова 1 по адресу 0x8048ff3: файл bed.c, строка 35. (gdb) run
Запуск
программы: binary_editor/bed
```

```
Точка останова 1, основная (argc=1, argv=0xbfa3e1f4) в bed.c:35
35          cfilesize = fstat.st_size;
(gdb)
```

Давайте проясним, что здесь произошло: GDB выполнил строки 30 через 34, но строка 35 еще не выполнена. Это может сбивать с толку, поскольку многие думают, что GDB отображает строку кода, которая была выполнена последней, когда на самом деле он показывает, какая строка кода будет выполнена . В этом случае GDB сообщает нам, что строка 35 — это следующая строка исходного кода для выполнения. Когда выполнение GDB достигает точки останова на строке 35, вы можете думать о GDB сижу и жду между строками 34 и 35 исходного кода.

Однако, как вы знаете, GDB работает с инструкциями машинного языка. ций, а не строк исходного кода, и может быть несколько строк машинного кода язык для одной строки кода. GDB может работать со строками исходного кода из-за дополнительной информации, включенной в исполняемый файл. В то время как это Сейчас этот факт может показаться не таким уж важным, но он будет иметь значение, когда мы обсудите пошаговое выполнение вашей программы в этой главе.

2.3 Отслеживание контрольных точек

Каждая точка останова (включая точки останова, точки наблюдения и точки перехвата) созданному вами присваивается уникальный целочисленный идентификатор, начинающийся с 1. Этот идентификатор используется для выполнения различных операций над точкой останова. Отладчик также включает в себя средство для перечисления всех ваших точек останова и их свойств.

2.3.1 Точка останова Спискив ГБД

Когда вы создаете точку останова, GDB сообщает вам присвоенный ей номер. например, точка останова, установленная в этом примере

```
(gdb) перерыв основной
Точка останова 2 по адресу 0x8048824: файл efh.c, строка 16.
```

был присвоен номер 2. Если вы когда-нибудь забудете, какой номер был присвоен какую именно точку останова вы можете вспомнить с помощью команды info breakpoints :

(gdb) информация о контрольных точках			
Num	Type	Disp Enb Address	Что
1	точка останова	сохраняет у 0x08048846	в Initialize_Game в efh.c:26
2	точки останова	сохраняют у 0x08048824	в main в efh.c:16
		точка останова уже достигнута 1 раз	
3	hw watchpoint	keep y 4 catch	уровень efh
		fork keep y	

Мы увидим, что эти идентификаторы используются для выполнения различных операций. на контрольных точках. Чтобы сделать это более конкретным, вот очень быстрый пример в заказ.

В предыдущем разделе вы видели команду удаления . Вы можете удалить точку останова 1, точку наблюдения 3 и точку перехвата 4 с помощью команды удаления с идентификаторами этих точек останова:

(gdb) удалить 1 3 4

В следующем выпуске вы увидите множество других вариантов использования идентификаторов точек останова. разделы.

2.3.2 Точка останова Спискив ДДД

Пользователи DDD в основном выполняют операции по управлению точками останова с помощью интерфейс «куаки и щелкни», поэтому идентификаторы точек останова менее важны для

Пользователи DDD, чем пользователи GDB. Выбор Источник | Точки останова

откройте окно «Точки останова и точки наблюдения», в котором перечислены все ваши точки останова, как показано на рисунке 2-1.

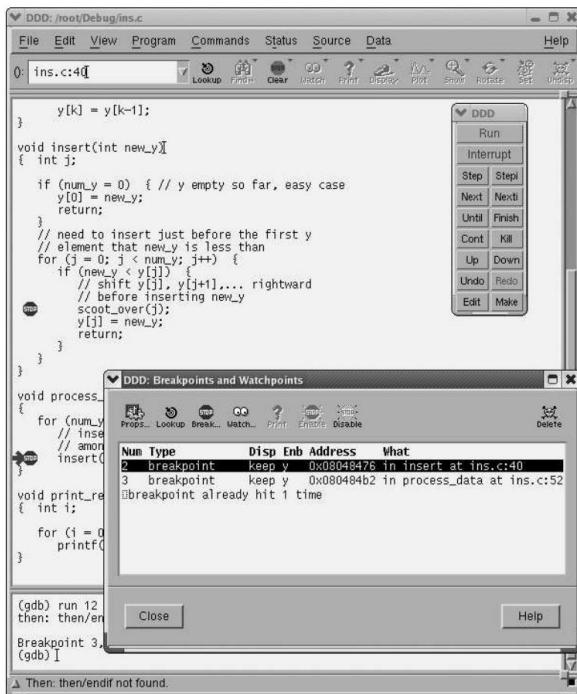


Рисунок2-1: Просмотр контролльные точки ДДД

Однако помните, что DDD позволяет вам использовать командный интерфейс GDB, а также предоставленный GUI. В некоторых случаях GDB предоставляет точку останова операции, которые недоступны через графический интерфейс DDD, но пользователь DDD может получить доступ к этим специальным операциям GDB через DDD Console. В таких случаях, Идентификаторы точек останова по-прежнему могут быть полезны пользователям DDD.

Обратите внимание, что вы можете оставить окно точек останова и точек наблюдения открытым. при желании можно перетаскивать его в удобную часть экрана.

2.3.3 Точка останова Списки в Eclipse

Перспектива Debug включает в себя представление Breakpoints. На рисунке 2-2, например, вы видите, что в настоящее время у вас есть две точки останова, на строках 30 и 52 файла ins.c.

Вы можете щелкнуть правой кнопкой мыши по записи любой точки останова, чтобы проверить или изменить ее. его свойства. Кроме того, двойной щелчок по записи приведет к фокусировке Окно исходного файла перемещается к этой точке останова.

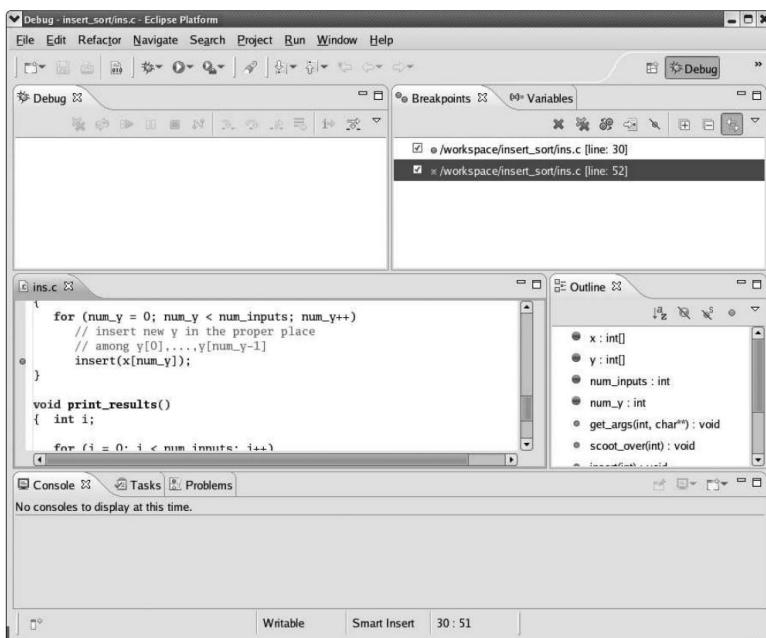


Рисунок2-2: Просмотр контрольные точки ^B Затмение

2.4 Установка точек останова

Инструменты отладки обычно предлагают различные механизмы, с помощью которых можно настроить контрольные точки.

2.4.1 Установка точки останова ГБД

Вы узнали, что GDB создает иллюзию выполнения программы строка за строкой. исходного кода. Вы также узнали, что для того, чтобы сделать что-то действительно полезное с GDB, вам нужно указать ему, где приостановить выполнение, чтобы вы могли выполнять отладочные действия с помощью командной строки GDB. В этом разделе

В ходе работы вы узнаете, как устанавливать точки останова, сообщающие GDB, где следует остановиться в исходном коде.

Существует много разных способов указать точку останова в GDB; вот некоторые из наиболее распространенных методов:

`перерыв функция`

Установите точку останова на входе (первая исполняемая строка) функции `function()`.

Пример этого вы видели в разделе 2.3.1; команда

(gdb) `перерыв основной`

устанавливает точку останова на входе в `main()`.

`перерыв номер_строки`

Установите точку останова на строке `line_number` текущего активного файла исходного кода. Для многофайловых программ это либо файл, содержимое которого вы последний раз просматривали с помощью команды `list`, либо файл, содержащий `main()`. Вы видели пример этого в Разделе 2.2; (gdb) `break 35`

который устанавливает точку останова на строке 35 в файле `bed.c`.

`перерыв имя_файла:номер_строки`

Установите точку останова на строке `line_number` файла исходного кода `filename`. Если `filename` отсутствует в вашем текущем рабочем каталоге, вы можете указать относительный или полный путь, чтобы помочь GDB найти файл, например:

(gdb) `перерыв источник/bed.c:35`

`перерыв имя_файла:функция`

Установите точку останова на входе функции `function()` в файле `filename`.

Использование этой формы может потребоваться для перегруженных функций или для программ, которые используют одноименные статические функции, например:

(gdb) `break bed.c:parseArguments`

Как мы увидим, когда установлена точка останова, она остается в силе до тех пор, пока вы ее не удалите, не отключите или не выйдете из GDB. Однако временная точка останова — это точка останова, которая автоматически удаляется после первого достижения. Временная точка останова устанавливается с помощью команды `tbreak`, которая принимает тот же тип аргументов, что и `break`. Например, `tbreak foo.c:10` устанавливает временную точку останова на строке 10 файла `foo.c`.

Необходимо сделать комментарий по поводу функций с одинаковыми именами. C++ позволяет перегружать функции (определять функции с одинаковыми именами).

Даже С позволяет вам сделать это, если вы используете статический квалификатор для объявления функций с файловой областью действия. Использование функции `break` установит точку останова на всех функциях с одинаковым именем. Если вы хотите установить точку останова на определенном экземпляре функции, вам нужно быть недвусмысленным, например, указав номер строки в файле исходного кода в вашей команде `break`.

Место, в котором GDB фактически устанавливает точку останова, может отличаться от того места, где вы просили его разместить. Это может быть несколько несогласованно

Другие аргументы перерыва

Существуют и другие, возможно, менее используемые аргументы, которые можно использовать с командой `break`.

- С помощью `break +offset` или `break -offset` вы можете установить точку останова смещает строки вперед или назад от исполняемой строки исходного кода в текущем выбранном стековом кадре.
- Форма

`break *address` может использоваться для установки точки останова по адресу виртуальной памяти. Это может быть необходимо для разделов программы, которые не имеют отладочной информации, например, когда исходный код недоступен или для разделяемых библиотек.

сертификация для новичков в GDB, поэтому давайте рассмотрим короткий пример, демонстрирующий эту странность. Рассмотрим следующую короткую программу, `test1.c`:

```

1 целочисленный основной (пустота)
2 {
3     целое
4     число i; i = 3;
5
6     возврат 0;
7 }
```

Скомпилируйте эту программу без оптимизации и попробуйте установить точку останова на входе в `main()`. Вы могли бы подумать, что точка останова будет установлена в верхней части функции — либо на строке 1, либо на строке 2, либо на строке 3. Это были бы хорошие догадки о местоположении точки останова, но они неверны. На самом деле точка останова установлена на строке 4.

```

$ gcc -g3 -Wall -Wextra -o test1 test1.c $ gdb test1
(gdb) break
main Точка останова
1 по адресу 0x6: файл test1.c, строка 4.
```

Строка 4 едва ли является первой строкой `main()`, так что же произошло? Как вы могли догадаться, одна из проблем заключается в том, что эта строка является исполняемой. Вспомните, что GDB на самом деле работает с инструкциями машинного языка, но с помощью магии расширенной таблицы символов GDB создает иллюзию работы со строками исходного кода. Обычно этот факт не так уж важен, но это ситуация, в которой он становится важным. На самом деле, объявление и действительно генерирует машинные

Другие типы точек останова

Кроме того, есть еще несколько команд `break-style`, которые в этой книге не будут подробно рассматриваться, поскольку они довольно специализированы.

Команда `hbreak` устанавливает аппаратную точку останова. Это точка останова, которую можно установить в ячейке памяти, не меняя ее содержимое. Для этого требуется аппаратная поддержка, и она в основном полезна для отладки EEPROM/ROM. Кроме того, есть команда `thbreak`, которая устанавливает временную аппаратную точку останова. Команды `hbreak` и `thbreak` принимают те же аргументы, что и `break` и `tbreak`.

Команда `rbreak` берет регулярное выражение в стиле `grep` и устанавливает точку останова на входе любой функции, которая соответствует регулярному выражению. Есть две вещи, о которых следует знать при использовании `rbreak`. Во-первых, важно помнить, что `rbreak` использует регулярные выражения в стиле `grep`, а не регулярные выражения в стиле Perl или подстановку в стиле shell. Это означает, например, что `rbreak func*` установит точки останова на функциях с именами `func` и `func()`, но не установит точку останова на `function()`. Во-вторых, подразумевается `.*` до и после вашего аргумента `rbreak`, поэтому, если вы не хотите, чтобы `rbreak func` устанавливал точку останова на `afunc()`, вам следует использовать `rbreak ^func`.

код, но это не тот код, который GDB считает полезным для наших целей отладки.¹ Поэтому, когда вы сказали GDB прерваться в начале `main()`, он установил точку останова на строке 4.

Проблема может усугубиться, если вы скомпилируете программу с оптимизацией. Давайте посмотрим. Перекомпилируйте программу с включенной оптимизацией:

```
$ gcc -O9 -g3 -Wall -Wextra -o test1 test1.c $ gdb
test1 (gdb)
break main Точка
останова 1 по адресу 0x3: файл test1.c, строка 6.
```

Мы попросили поставить точку останова в начале `main()`, но GDB поставил ее на последней строке `main()`. Что же произошло? Ответ тот же, что и раньше, но GCC взял на себя более активную роль. При включенной оптимизации GCC заметил, что хотя `i` было присвоено значение, оно никогда не использовалось. Поэтому в попытке сгенерировать более эффективный код GCC просто оптимизировал строки 3 и 4, исключив их существование. GCC никогда не генерировал машинные инструкции для этих строк. Таким образом, первая строка исходного кода, которая генерировала машинные инструкции,

¹ Просмотреть машинный код, сгенерированный GCC, можно с помощью опции `-S`.

Точки захвата

Программисты C++ захотят проверить команду `catch`, которая устанавливает точки перехвата. Точки перехвата похожи на точки останова, но могут быть вызваны разными вещами, такими как выброшенные исключения, перехваченные исключения, сигналы, вызовы `fork()`, загрузка и выгрузка библиотек и многие другие события.

В этой книге мы рассмотрим точки останова и наблюдения, но отшлем читателя к документации GDB за дополнительной информацией о точках перехвата.

случайно оказывается последней строкой `main()`. Это одна из причин, по которой вам никогда не следует оптимизировать код, пока вы не закончите его отладку.²

Результатом всего этого является то, что если вы обнаружите, что установка точки останова не создает точку останова именно там, где вы ожидаете, теперь вы знаете, почему. Не удивляйтесь.

По смежному вопросу мы хотели бы упомянуть, что происходит, когда на одной строке исходного кода находится более одной точки останова. Когда GDB останавливается на строке исходного кода с более чем одной точкой останова, он останавливается там только один раз. Другими словами, после того, как он достигнет этой строки кода, если вы возобновите выполнение, другие точки останова, которые оказались на той же строке, будут проигнорированы. Фактически, GDB отслеживает, какая точка останова «запустила» остановку выполнения программы. В строке кода с несколькими точками останова, которая запускает останов, будет точкой останова с наименьшим идентификатором.

2.4.2 Параметр Точки останова В ДДД

Чтобы установить точки останова с помощью DDD, найдите строку кода, в которой вы хотите установить точку останова в окне исходного кода. Поместите курсор на любое пустое место в этой строке и щелкните правой кнопкой мыши, чтобы открыть всплывающее меню. Перетащите мышь вниз, пока не будет выделена опция `Set Breakpoint`, а затем отпустите кнопку мыши. Вы должны увидеть красный знак остановки рядом со строкой кода, в которой вы установили точку останова. Если вы не делаете ничего особенного с точкой останова, например, не делаете ее условной (что будет обсуждаться в разделе 2.10), сокращением будет просто двойной щелчок по заданной строке.

Если вы экспериментировали с DDD, вы могли заметить, что при нажатии правой кнопки рядом со строкой кода всплывающее меню содержит пункт `Set Temporary Breakpoint`. Вот как вы устанавливаете временную точку останова (точку останова, которая исчезает после первого достижения) с помощью DDD, что вызывает команду GDB `tbreak`.

² На самом деле, некоторые отладчики могут действительно подавиться исполняемыми файлами, скомпилированными с включенной оптимизацией. GDB отличается тем, что является одним из немногих отладчиков, которые могут отлаживать оптимизированный код, но, как вы видели, результаты могут быть сомнительными.

Опять же, не забывайте, что DDD на самом деле является фронтеном GDB. Вы можете выполнить любые команды останова в стиле GDB в DDD с использованием окна консоли DDD.

Иногда это желательно; если у вас очень большая или многофайловая программа, может быть удобнее установить точку останова, используя семантику GDB.

На самом деле, иногда это необходимо, поскольку не все команды точек останова GDB можно вызвать из интерфейса DDD.

2.4.3 Параметр Точки останова в Eclipse

Чтобы установить точку останова на заданной строке в Eclipse, дважды щелкните по этой строке.

Появится символ точки останова, как показано, например, в строке

```
вставить(x[num_y]);
```

на рисунке 2-2.

Чтобы установить временную точку останова, щелкните строку, затем щелкните правой кнопкой мыши в исходном окне и выберите Run to Line. Обратите внимание, однако, что операция Run to Line работает только в том случае, если целевая строка находится в той же функции, что и ваша текущая позиция, и если вы не вышли из функции перед повторным входом и попаданием на эту строку.

2.5 Расширенный пример GDB

Это было много информации, поэтому краткий пример установки точек останова, которому вы можете следовать, оправдан. Рассмотрим следующий многофайловый код C:

main.c:

```
#include <stdio.h>
void swap(int *a, int *b);
целочисленный основной(void)
{
    целое i = 3;
    целое j = 5;

    printf("i: %d, j: %d\n", i, j); swap(&i,
    &j); printf("i:
    %d, j: %d\n", i, j);

    возврат 0;
}
```

swapper.c:

```
недействительный своп(целое *a, целое *b)
{
    int c = *a; *a
    = *b; *b
    = c;
}
```

Скомпилируйте этот код и запустите GDB для исполняемого файла:

```
$ gcc -g3 -Wall -Wextra -c main.c swapper.c $ gcc -o swap
main.o swapper.o $ gdb swap
```

ПРИМЕЧАНИЕ Это первый раз в этой книге, когда мы компилируем многофайловую программу на языке C, поэтому слово в порядке. Первая строка процесса компиляции (выше) создает два объектных файла, содержащих неразрешенный объектный код с отладочной информацией. Вторая строка связывает объектные файлы в исполняемый файл, содержащий всю отладочную информацию. Нет необходимости использовать переключатель GCC `-g` во время процесса связывания.

Установка точки останова в `main()` очень распространена при запуске отладки. сеанс. Это устанавливает точку останова на первой строке этой функции.³

```
(gdb) break main
Точка останова 1 по адресу 0x80483f6: файл main.c, строка 6.
```

Все приведенные ниже примеры устанавливают точку останова на первой строке функции `swap()`. Хотя они могут выглядеть по-разному, все они делают одно и то же: останавливаются в верхней части `swap()`.

```
(gdb) break swapper.c:1 Точка
останова 2 по адресу 0x8048454: файл swapper.c, строка 1. (gdb)
break swapper.c:swap Точка
останова 3 по адресу 0x804845a: файл swapper.c, строка 3. (gdb)
break swap

Примечание: точка останова 3 также установлена на компьютере с адресом 0x804845a.
Точка останова 4 по адресу 0x804845a: файл swapper.c, строка 3.
```

В любой момент времени GDB имеет (за исключением лучшего слова) фокус, на котором вы можете считать текущим «активным» файлом. Это означает, что если вы не квалифицируете свои команды, они выполняются над файлом, на котором находится фокус GDB. По умолчанию файл, на котором находится начальный фокус GDB, — это файл, содержащий функцию `main()`, но фокус меняется на другой файл, когда происходит любое из следующих действий:

- Вы применяете команду `list` к другому исходному файлу.
- Вы входите в код, находящийся в другом исходном файле.
- GDB достигает точки останова при выполнении кода в другом исходном файле.

Давайте рассмотрим пример. Хотя точки останова были установлены в `swap-per.c`, мы фактически не перечислили код из этого файла. Следовательно, фокус все еще тренируется на `main.c`. Вы можете проверить это, установив точку останова в строке 6. Когда

³ Помните, что точка останова может быть не совсем на первой строке `main()`, но она будет близко. В отличие от Однако в нашем предыдущем примере эта строка является исполняемой, поскольку она присваивает значение к я.

вы устанавливаете эту точку останова без имени файла, GDB установит точку останова в строка 6 текущего активного файла:

(gdb) перерыв 6

Точка останова 5 по адресу 0x8048404: файл main.c, строка 6.

Конечно же, main.c находится в фокусе: когда вы запускаете GDB, точка останова устанавливается только по номеру строки, установленному в файле, содержащем main(). Вы можете изменить фокус, перечислив код из swapper.c:

(gdb) список обмена

```
1 недействительный своп(int *a, int *b)
2 {
3     int c = *a;
4     *a = *b;
5     *b = c;
6 }
```

Давайте проверим, что swapper.c теперь имеет фокус, попробовав установить другой точка останова на строке 6:

(gdb) перерыв 6

Точка останова 6 по адресу 0x8048474: файл swapper.c, строка 6.

Да, точка останова была установлена на строке 6 файла swapper.c. Затем вы установите временную точку останова на строке 4 файла swapper.c:

(gdb) tbreak swapper.c:4

Точка останова 7 по адресу 0x8048462: файл swapper.c, строка 4.

Наконец, используйте команду info breakpoints , представленную в разделе 2.3.1, чтобы увидеть все установленные вами точки останова:

(gdb) информация о контрольных точках

Номер	Disp	Enb	Address	Что
Тип 1 точка			сохранить у 0x080483f6 в main в main.c:6	
останова 2 точка			сохранить у 0x08048454 в файле swap в swapper.c:1	
останова 3 точка			сохранить у 0x0804845a в swap в swapper.c:3	
останова 4 точка			сохранить у 0x0804845a в swap в swapper.c:3	
останова 5 точка			сохранить у 0x08048404 в main в main.c:9	
останова 6 точка			сохранить у 0x08048474 в swap в swapper.c:6	
останова 7 точка останова	del		у 0x08048462 в swap в swapper.c:4	

Гораздо позже, когда вы закончите сеанс GDB, используйте команду quit команда выхода из GDB:

```
(gdb) выйти
$
```

2.6 Сохранение контрольных точек

Мы сказали «гораздо позже» выше, чтобы подчеркнуть, что вам не следует выходить GDB во время сеанса отладки. Например, когда вы находите и исправляете одна ошибка, но другие ошибки остались, вам не следует выходить и снова входить в GDB использовать новую версию вашей программы. Это было бы лишней проблемой, и что еще важнее, вам придется заново вводить контрольные точки.

Если вы не выйдете из GDB при изменении и перекомпиляции кода, в следующий раз, когда вы дадите команду GDB «Выполнить», GDB почувствует, что ваш код изменены и автоматически перезагружены.

Однако учтите, что ваши контрольные точки могут «перемещаться». Например, рассмотрите следующая простая программа:

```
1 основной()
2 { int x,y;
3     x = 1;
4     y = 2;
5 }
```

Компилируем, входим в GDB и устанавливаем точку останова на строке 4:

```
(гдб) л
1      основной()
2      { int x,y;
3          x = 1;
4          y = 2;
5      }
(гдб) б 4
Точка останова 1 по адресу 0x804830b: файл ас, строка 4.
(гдб) р
Запуск программы: /usr/home/matloff/Tmp/tmp1/a.out
```

Точка останова 1, main () в ас:4

```
4          y = 2;
```

Все хорошо. Но предположим, что вы теперь добавляете исходную строку:

```
1 основной()
2 { int x,y;
3     x = 1;
4     x++;
5     y = 2;
6 }
```

Затем перекомпилируйте — опять же, помните, что вы не вышли из GDB — и снова введите команду запуска GDB :

```
(gdb) r
Отлаживаемая программа уже запущена.
Начать с начала? (у или n) у `/usr/home/matloff/
Tmp/tmp1/a.out' изменился; перечитывание символов.
```

Запуск программы: /usr/home/matloff/Tmp/tmp1/a.out

```
Точка останова 1, main () в ac:4
4           x++;
```

GDB перезагрузил новый код, но точка останова, по-видимому, сместилась с оператора

```
y = 2;
```

к

```
x++;
```

Если вы посмотрите внимательнее, то увидите, что точка останова на самом деле не установлена. перемещено вообще; оно было на строке 4, и оно все еще на строке 4. Но эта строка больше не содержит оператора, на котором вы изначально установили точку останова. Таким образом, вам нужно будет переместить точку останова, удалив ее и установив новую. (В DDD это можно сделать гораздо проще; см. раздел 2.7.5.)

В конце концов, ваш текущий сеанс отладки закончится, например, потому что пришло время поесть, поспать или отдохнуть. Если вы обычно не держите компьютер включенным непрерывно, вам нужно будет выйти из отладчика. Есть ли способ сохранить ваши точки останова?

Для GDB и DDD ответ — да, в некоторой степени. Вы можете разместить свои точки останова в файле запуска .gdbinit в каталоге, где находится ваш исходный код (или в каталоге, из которого вы вызываете GDB).

Если вы работаете в Eclipse, вам повезло, потому что все ваши точки останова будут автоматически сохраняться и восстанавливаться в следующем сеансе Eclipse.

2.7 Удаление и отключение точек останова

В ходе сеанса отладки вы можете обнаружить, что точка останова изжила себя. Если вы уверены, что точка останова больше не понадобится, вы можете удалить ее.

Также может быть так, что вы думаете, что точка останова может быть полезна вам позже во время сеанса отладки. Возможно, вы предпочтете не удалять точку останова, а вместо этого сделать так, чтобы отладчик на время пропускал остановы в этой точке вашего кода. Это называется отключением точки останова. Вы можете снова включить ее позже, если/когда она снова станет полезной.

В этом разделе рассматривается удаление и отключение точек останова. Все, что упоминается-то же самое относится и к точкам наблюдения.

2.7.1 Удаление Точки остановав ГБД

Если рассматриваемая точка останова действительно больше не нужна (возможно, эта конкретная ошибка была исправлена!), то вы можете удалить эту точку останова. Есть два

Команды, которые используются для удаления точек останова в GDB. Команда `delete` используется для удаления точек останова на основе их идентификатора, а команда `clear`

Команда используется для удаления точек останова с использованием того же синтаксиса, который используется для создания точек останова, как обсуждалось в разделе 2.4.1.

`удалить список_точек_останова`

Удаляет контрольные точки, используя их числовые идентификаторы (которые были объяснены в разделе 2.3). Это может быть одно число, например, `delete 2`, которое удаляет вторую точку останова или список чисел, например, `удалить 2 4`, который удаляет вторую и четвертую точки останова.

`удалить`

Удаляет все точки останова. GDB попросит вас подтвердить эту операцию, если вы не выполните команду `set confirm off`, которую также можно поместить в ваш файл запуска `.gdbinit`.

`прозрачный`

Очищает точку останова на следующей инструкции, которую выполнит GDB. Это Метод полезен, когда вы хотите удалить точку останова, установленную GDB. только что дошел.

`прозрачный функция`

`прозрачный имя_файла:функция`

`прозрачный номер_строки`

`прозрачный имя_файла:номер_строки`

Они очищают точку останова в зависимости от ее местоположения и работают аналогично коллеги по перерыву.

Например, предположим, что вы установили точку останова на входе `foo()` с помощью:

(gdb) перерыв фу

Точка останова 2 по адресу 0x804843a: файл test.c, строка 22.

Вы можете удалить эту точку останова либо с помощью

(gdb) очистить foo

Удален остановочный пункт 2

или с

(gdb) удалить 2

Удален остановочный пункт 2

2.7.2 Отключение Точки останова в ГБД

Каждая точка останова может быть включена или отключена. GDB приостановит программу выполнение только при достижении включенной точки останова; отключенные точки останова игнорируются. По умолчанию точки останова начинают свою жизнь как включенные.

Зачем вам может понадобиться отключить точки останова? В ходе сеанса отладки вы можете собрать большое количество точек останова. Цикл For структуры или функции, которые часто повторяются, могут быть крайне неудобны для GDB так часто ломаться. Если вы хотите сохранить точки останова для дальнейшего использования но не хотите, чтобы GDB останавливал выполнение на данный момент, вы можете отключить и включить их позже.

Вы можете отключить точку останова с помощью команды `disable breakpoint-list`. и включите точку останова с помощью команды `enable breakpoint-list`, где `breakpoint-list` — это список из одного или нескольких идентификаторов точек останова, разделенных пробелами. Например,

(gdb) отключить 3

отключит третью точку останова. Аналогично,

(gdb) включить 1 5

включит первую и пятую точки останова.

Выполнение команды `disable` без аргументов отключит все существующие точки останова. Аналогично, команда `enable` без аргументов отключит все существующие точки останова. включить все существующие точки останова.

Также есть команда `enable once`, которая активирует точку останова... отключается после того, как в следующий раз это заставит GDB приостановить выполнение.

Синтаксис:

включить один раз `breakpoint-list`

Например, включение однократного 3 приведет к отключению точки останова 3. в следующий раз это заставит GDB остановить выполнение вашей программы. Это очень похожа на команду `tbreak`, но она отключает, а не удаляет, когда обнаружена точка останова.

2.7.3 Удаление и Отключение Точки останова в ДДД

Удаление точек останова в DDD так же просто, как и их установка. Поместите курсор на красный знак остановки и щелкните правой кнопкой мыши, как если бы вы устанавливали Breakpoint. Одним из вариантов во всплывающем меню будет Delete Break-point. Удерживайте правую кнопку мыши и перетащите мышь вниз, пока эта опция выделена. Затем отпустите кнопку, и вы увидите красный Знак «Стоп» исчезнет, указывая на то, что точка останова была удалена.

Отключение точек останова с помощью DDD очень похоже на их удаление. Щелкните правой кнопкой мыши и удерживайте красный значок остановки, затем выберите Отключить точку останова. Красный значок остановки

знак станет серым, указывая на то, что точка останова все еще существует, но отключена на данный момент.

Другой вариант — использовать окно DDD Breakpoints and Watchpoints, показанное на рисунке 2-1.

Вы можете щелкнуть запись breakpoint там, чтобы выделить и выберите Удалить, Отключить или Включить.

Фактически, вы можете выделить несколько записей в этом окне, перетащив наведите на них курсор, как показано на рисунке 2-3. Таким образом, вы можете удалить, отключить или включить несколько точек останова одновременно.

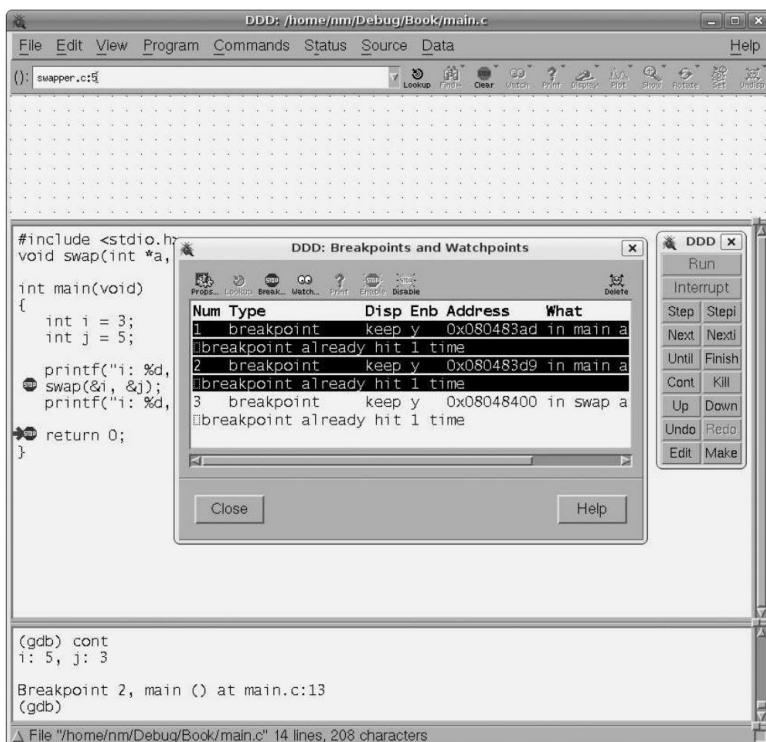


Рисунок2-3: Удаление/отключение/включение нескольких точек останова в ДДД

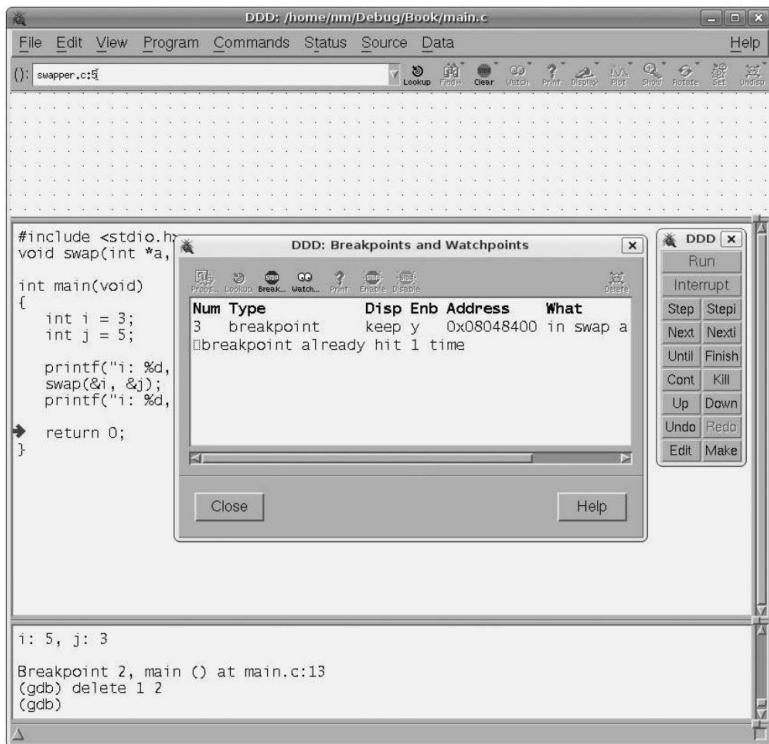
После нажатия кнопки «Удалить» в окне точек останова экран выглядит следующим образом:

Рисунок 2-4. Конечно же, две старые точки останова теперь исчезли.

2.7.4 Удаление и Отключение Точки останова в Eclipse

Как и в случае с DDD, вы можете удалить или отключить точку останова в Eclipse, щелкнув правой кнопкой мыши по символу точки останова в строке, о которой идет речь. Появится меню, показано на рисунке 2-5. Обратите внимание, что опция Toggle означает удаление точки останова, в то время как Disable/Enable означает очевидное.

Подобно окну точек останова DDD, в Eclipse есть свой вид точек останова, который вы можете видеть в верхней правой части рисунка 2-5. В отличие от DDD случай, в котором нам нужно было запросить Breakpoints и Watchpoints win-



Фигура 2-4: точки останова удалены

Теперь представление точек останова Eclipse автоматически отображается в перспективе отладки. (Однако вы можете скрыть его, если у вас мало места на экране, нажав на X в его правом углу. Если вы захотите вернуть его позже, выберите Окно | Показать представления | Точки останова.)

Одним из приятных аспектов представления точек останова Eclipse является то, что вы можете щелкнуть по значку двойной X (Удалить все точки останова). Необходимость в этом возникает чаще, чем вы могли бы предположить. В некоторые моменты долгого сеанса отладки вы можете оказаться, что ни одна из установленных вами ранее точек останова теперь не полезна, и, таким образом, хочется удалить их все.

2.7.5 "Движущийся" Точки останова в DDD

У DDD есть еще одна действительно полезная функция: перетаскивание точек останова. Щелкните левой кнопкой мыши и удерживайте знак остановки точки останова в окне исходного кода. Пока вы держите левую кнопку, вы можете перетащить точку останова в другое место в вашем исходном коде. Что происходит «за кулисами», так это то, что DDD удаляет исходную точку останова и устанавливает новую точку останова с теми же атрибутами. В результате вы обнаружите, что новая точка останова полностью эквивалентна старой точке останова, за исключением ее числового идентификатора. Конечно, вы можете то же самое сделать и с GDB, но DDD ускоряет процесс.

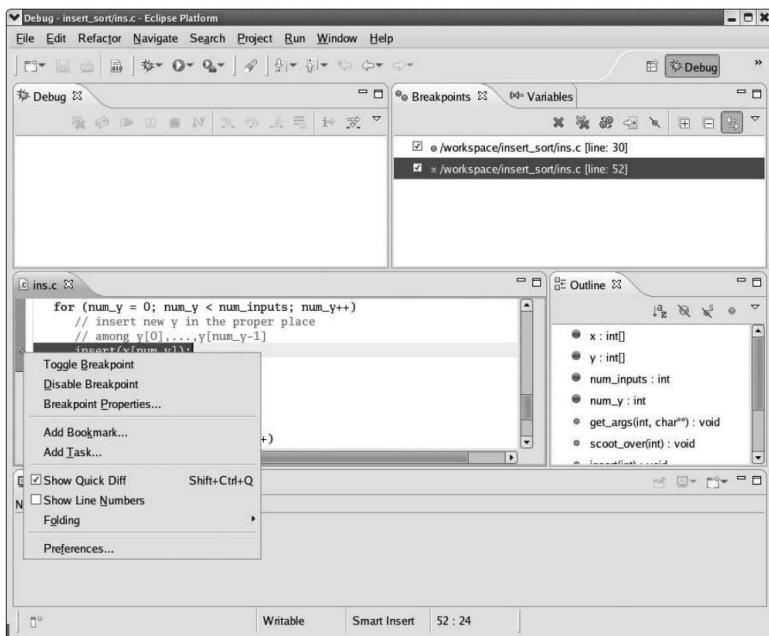


Рисунок2-5: Удаление/отключение точек останова в Затмение

Рисунок 2-6 иллюстрирует это — снимок, сделанный в разгар
Операция breakpoint-move. На строке была breakpoint

если (новый_y < y[i]) {

которые мы хотели перенести на линию

scoot_over();

Мы нажали на знак «Стоп» на старой линии и перетащили его на новую.
линия. На рисунке мы еще не отпустили кнопку мыши, поэтому оригинальный знак
остановки все еще был там, а на новом появился «пустой» знак остановки
линия. После того, как мы отпустили кнопку мыши, знак остановки на старой линии исчез, а
на новой линии заполнился.

Почему это так полезно? Прежде всего, если вы находитесь в ситуации, в которой вы
хотите удалить одну точку останова и добавить другую, это делает обе операции
одним махом. Но что еще важнее, как было отмечено ранее в разделе 2.6,
Когда вы добавляете или удаляете строки исходного кода, номера некоторых оставшихся строк
изменяются, тем самым «сдвигая» существующие точки останова на строки, на которых вы
не собирались его иметь. В GDB это требует от вас выполнения действий удаления точки
останова и создания новой точки останова в каждой из затронутых точек останова.
Это утомительно, особенно если у некоторых из них есть какие-то условия.

Но в DDD вы можете просто перетащить значок знака «Стоп» в новое место — очень красиво и удобно.

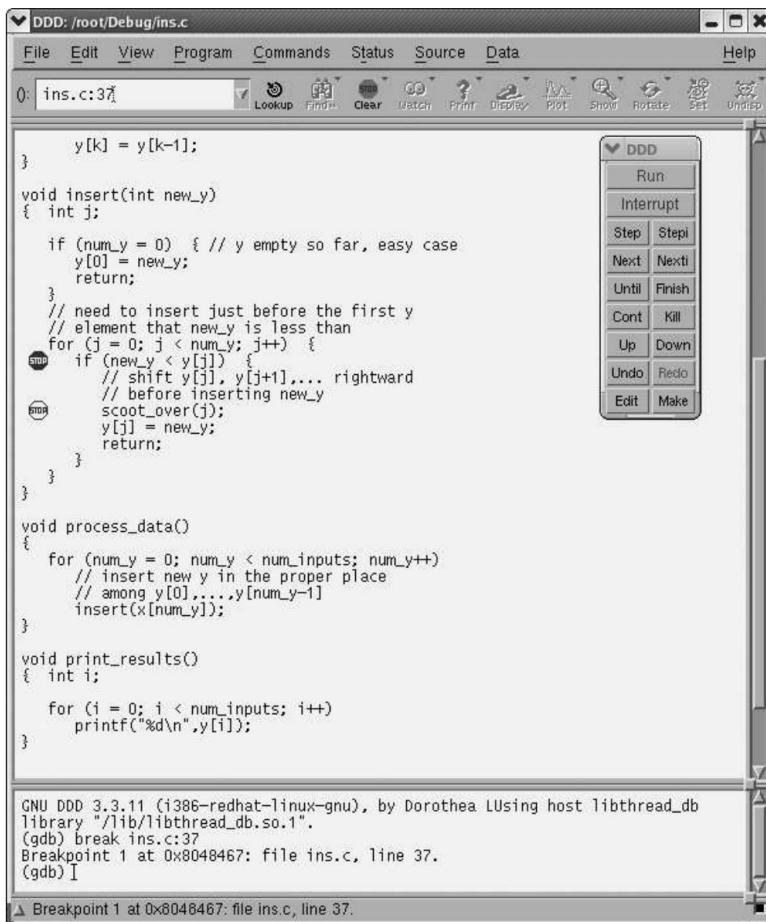


Рисунок2-6: "Движущийся" а точка останова В ДДД

2.7.6 Отмена/Повторное выполнение Точка останова Действия в ДДД

Одной из действительно полезных функций DDD является функция «Отменить/Повторить», доступ к которой осуществляется нажатием кнопки «Изменить».

В качестве примера рассмотрим ситуацию на рисунках 2-3 и 2-4. Предположим, вы внезапно осознаете, что вы не хотели удалять эти две точки останова — может быть, вы просто хотели отключить их. Вы можете выбрать Редактировать | Отменить удаление, показано на рисунке 2-7. (Вы также можете нажать «Отменить» в командной строке, но прохождение Edit имеет то преимущество, что DDD напомнит нам, что будет отменено.)

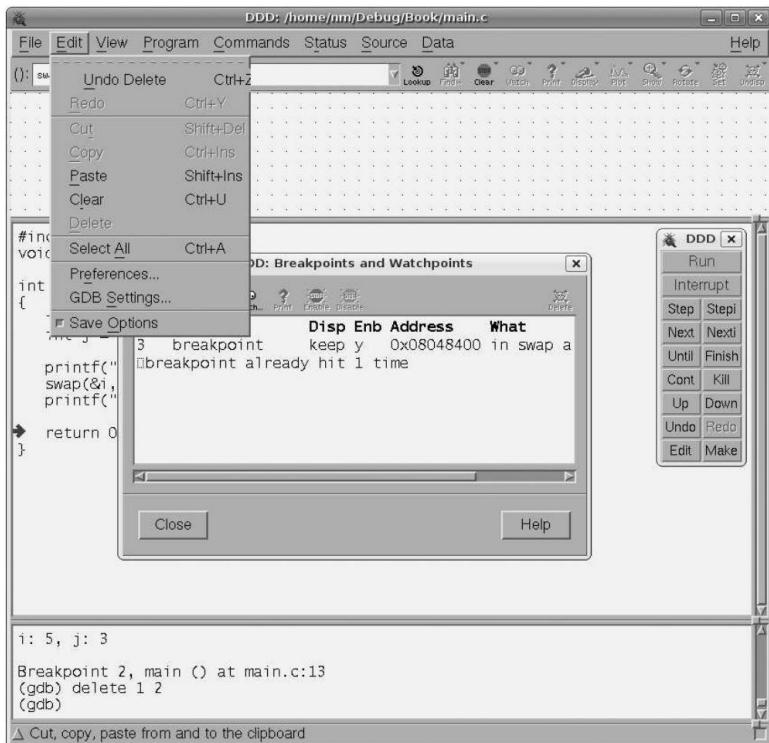


Рисунок 2-7: А вставить два контрольные точки в ДДД

2.8 Подробнее о просмотре атрибутов точек останова

Каждая точка останова имеет различные атрибуты — номер строки, наложенное на нее условие (если таковое имеется), текущий статус включения/выключения и т. д. Мы показали, в разделе 2.3 немного об отслеживании этих атрибутов, а теперь перейдем более подробно.

2.8.1 ГБД

Как вы видели в разделе 2.3.1, каждой созданной вами точке останова присваивается уникальный целочисленный идентификатор. Первая установленная вами точка останова присваивается значению «1», и каждый новая точка останова затем назначается целое число на единицу больше, чем ранее назначенный идентификатор. Каждая точка останова также имеет ряд атрибутов которые контролируют и настраивают его поведение. Используя уникальные идентификаторы, вы можете настройте атрибуты каждой точки останова по отдельности.

Вы можете использовать команду `info breakpoints` (сокращенно `i b`) для получения списка всех установленных вами точек останова вместе с их атрибутами.

Вывод информационных точек останова будет выглядеть примерно так:

(gdb) информация о контрольных точках

Тип номера

Disp Enb Адрес

Что

```
1 точка останова удерживает у 0x08048404 в main в int_swap.c:9 точка останова уже
сработала 1 раз 2 точка останова
удерживает п 0x08048447 в main в int_swap.c:14 3 точка останова удерживает у
0x08048460 в swap в int_swap.c:20 точка останова уже сработала 1 раз
```

4 hw watchpoint keep у

прилавок

Давайте подробно рассмотрим вывод информационных точек останова :

Идентификатор

(Num): уникальный идентификатор точки останова.

Тип (Type): В

этом поле указывается, является ли точка останова точкой останова, точкой наблюдения или точкой перехвата.

Disposition (Disp): Каждая

точка останова имеет disposition, который указывает, что произойдет с точкой останова после того, как она в следующий раз заставит GDB приостановить выполнение программы. Существует три возможных disposition: keep Точка останова останется

неизменной после того, как она будет достигнута в следующий раз.

Это стандартное расположение вновь созданных точек останова. del Точка

останова будет удалена после следующего ее достижения.

Это расположение назначается любой точке останова, которую вы создаете с помощью команды tbreak (см. раздел 2.4.1). dis

Точка останова будет отключена при следующем ее достижении. Это устанавливается с помощью команды enable once (см. раздел 2.7.2).

Состояние включения

(Enb): в этом поле указывается, включена или отключена в данный момент точка останова.

Адрес (Address): Это

место в памяти, где установлена точка останова. Это в основном будет полезно программистам на языке ассемблера или людям, пытающимся отладить исполняемый файл, который не был скомпилирован с расширенной таблицей символов.

Местоположение

(Что): Как уже обсуждалось, каждая точка останова находится на определенной строке в исходном коде. Поле Что показывает номер строки и имя файла местоположения точки останова.

Для точек наблюдения это поле показывает, какая переменная отслеживается.

Это имеет смысл, поскольку переменная на самом деле представляет собой адрес памяти с именем, а адрес памяти — это местоположение.

Как вы можете видеть, в дополнение к перечислению всех точек останова и их атрибутов, команда ib также сообщает вам, сколько раз конкретная точка останова заставляла GDB останавливать выполнение программы до сих пор. Если, например, у вас есть точка останова внутри цикла, она сразу скажет вам, как

На данный момент выполнено много итераций цикла, что может быть очень полезный.

2.8.2 ДДД

Как вы видели на рисунке 2-1, окно точек останова и точек наблюдения DDD предоставляет ту же информацию, что и команда GDB info breakpoints . Однако оно удобнее, чем GDB, так как вы можете отображать это окно постоянно (т.е. в стороне от экрана), что позволяет избежать необходимости отдавать команду каждый раз время, когда вы хотите просмотреть свои точки останова.

2.8.3 Затмение

Опять же, как было показано ранее (рисунок 2-2), вид точек останова Eclipse постоянно отображает ваши точки останова и их свойства. Eclipse немного менее информативен чем DDD здесь, в том смысле, что он не сообщает вам, сколько раз была установлена точка останова пока не было поражено (этой информации даже нет в окне «Свойства»).

2.9 Возобновление выполнения

Знание того, как указать отладчику, где или когда следует приостановить выполнение вашей программы важно, но знание того, как дать ей команду возобновить выполнение, так же важно. В конце концов, проверка ваших переменных может быть не достаточно. Иногда вам нужно знать, как значения переменных взаимодействуют с остальной частью кода.

Вспомните принцип подтверждения из главы 1: вы продолжаете конфиденциально, что определенные переменные имеют те значения, которые вы думаете, что они имеют, пока вы не столкнетесь с той, которая не соответствует вашим ожиданиям. Эта неудача будет затем подсказкой о вероятном местоположении вашего бага. Но обычно сбой не будет происходить до тех пор, пока вы не приостановите и не возобновите выполнение в нескольких точках останова (или несколько раз в одной и той же точке останова). Таким образом, возобновление выполнения в точке останова так же важно, как и установка самой точки останова. Вот почему инструмент отладки обычно имеет довольно богатый набор методов для возобновления исполнения.

Существует три класса методов возобновления выполнения. Первый включает «пошаговое выполнение» вашей программы с шагом и следующим, выполнением только следующую строку кода и затем снова паузу. Второй состоит из с помощью continue , что заставляет GDB безоговорочно возобновить выполнение программы, пока не достигнет другой точки останова или программа не завершится. Последний класс методов включает условия: возобновление с командами finish или until . В этом случае GDB возобновит выполнение и программа запустится пока не будет выполнено некоторое предопределенное условие (например, не будет достигнут конец функции), не будет достигнута другая точка останова или пока программа не завершится.

Мы рассмотрим каждый метод возобновления выполнения по очереди для GDB, а затем показать, как выполнять такие операции в DDD и Eclipse.

2.9.1 В ГБД

Мы начнем этот раздел с обсуждения различных способов возобновления выполнения после остановки GDB в точке останова.

2.9.1.1 Пошаговое выполнение с шагом и следующим

После того, как GDB останавливается на точке останова, команды next (сокращенно n) и step (сокращенно s) используются для пошагового выполнения вашего кода. После того, как срабатывает точка останова и GDB останавливается, вы можете использовать next и step для выполнить только следующую строку кода. После того, как строка будет выполнена, GDB снова сделайте паузу и дайте приглашение на ввод команды. Давайте посмотрим на это в действии. Рассмотрим программу swapflaw.c:

```

1 /* swapflaw.c: Некорректная функция, которая меняет местами два целых числа. */
2 #include <stdio.h>
3 void swap(int a, int b);
4
5 целочисленный основной(пустой)
6 {
7     целое число i = 4;
8     int j = 6;
9
10    printf("i: %d, j: %d\n", i, j);
11    поменять местами(i, j);
12    printf("i: %d, j: %d\n", i, j);
13
14    возврат 0;
15 }
16
17 недействительный своп (целое а, целое б)
18 {
19     целочисленный с = а;
20     а = 6;
21     б = с;
22 }
```

Листинг 2-1: swapflaw.c

Мы установим точку останова на входе в main() и запустим программу в ГБД.

```

$ gcc -g3 -Wall -Wextra -o swapflaw swapflaw.c
$ gdb swapflaw
(gdb) перерыв основной
Точка останова 1 по адресу 0x80483f6: файл swapflaw.c, строка 7.
(gdb) запустить
Запуск программы: swapflaw
```

Точка останова 1, main() в swapflaw.c:7

7 целое число i = 4;

GDB теперь находится на строке 7 программы, что означает, что строка 7 не была пока не выполнено. Мы можем использовать следующую команду, чтобы выполнить только эту строку кода, оставляя нас перед строкой 8:

(gdb) следующий
8 int j = 6;

Мы используем step для выполнения следующей строки кода, строки 8, которая переместит нас к строка 10.

(gdb) шаг
10 printf("i: %d, j: %d\n", i, j);

Мы видим, что и next , и step выполняют следующую строку кода. Так что большой вопрос: «Чем отличаются эти команды?» Они обе, кажется, выполняют следующую строку кода. Разница между этими двумя командами в том, как они обрабатывают вызовы функций: next выполнит функцию, не останавливаясь внутри нее, а затем остановится на первом операторе, следующем за вызовом. step, он с другой стороны, останавливается на первом операторе внутри функции.

Вызов swap() происходит в строке 11. Давайте посмотрим на эффект следующего и шагайте бок о бок.

Используя шаг:

(gdb) шаг
я: 4, ж: 6
11 поменять местами(i, j);

(gdb) шаг
своп (a=4, b=6) в swapflaw.c:19
19 целочисленный c = a;

(gdb) шаг
20 a = b;

(gdb) шаг
21 б = с;

(гдб) шаг
22 }

(gdb) шаг
main() в swapflaw.c:12
12 printf("i: %d, j: %d\n", i, j);

(gdb) шаг
я: 4, ж: 6
14 возврат 0;

Используя следующее:

(gdb) следующий
я: 4, ж: 6
11 поменять местами(i, j);

(gdb) следующий
12 printf("i: %d, j: %d\n", i, j);

(gdb) следующий
я: 4, ж: 6
14 возврат 0;

Команда step работает так, как вы могли бы ожидать. Она выполнила printf() в строке 10, затем вызов swap() в строке 11⁴, а затем начала выполнять строки кода внутри swap(). Это называется пошаговым выполнением функции.

После того, как мы пройдем все строки swap(), шаг возвращает нас к main().

Напротив, кажется, что next никогда не покидал main(). Это главное отличие разницы между двумя командами. next рассматривает вызов функции как одну строку кода и выполняет всю функцию за одну операцию, что называется перешагиванием через функцию.

Однако не обманывайтесь; может показаться, что next пропустил тело swap(), но на самом деле он ничего не «перешагнул». GDB молча выполнил каждую строку swap(), не показывая нам деталей (хотя он показывает любой вывод на экран, который swap() может вывести) и не предлагая нам выполнить отдельные строки из функции.

Разница между входом в функцию (что делает шаг) и выходом за пределы функции (что делает следующий) настолько важна, что, рискуя усложнить изложение, мы продемонстрируем разницу между следующим и шагом с помощью диаграммы, которая показывает выполнение программы с помощью стрелок.

Рисунок 2-8 иллюстрирует поведение команды step . Представьте, что программа приостановлена на первом операторе printf() . Рисунок показывает, куда нас приведет каждый оператор step :

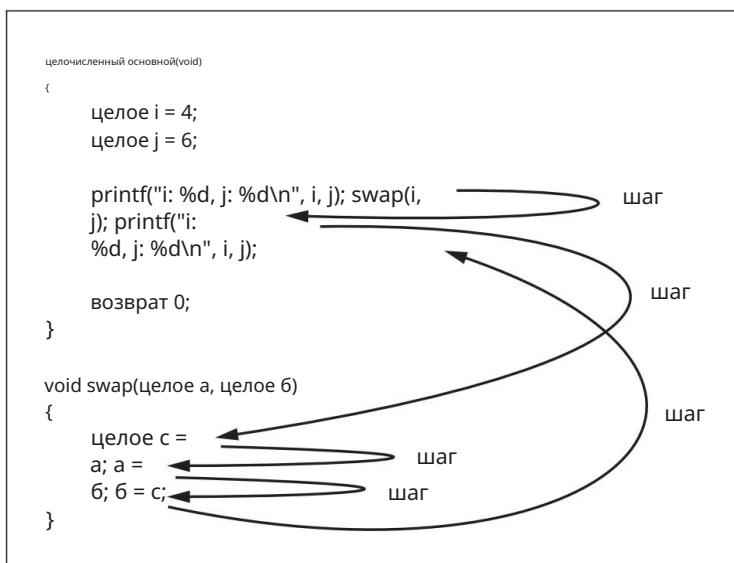


Рисунок 2-8: Шаг шаги в функция.

Рисунок 2-9 иллюстрирует то же самое, но с использованием next .

⁴ Вы можете удивиться, почему step не перенес вас на первую строку функции printf(). Причина в том, что GDB не останавливается в коде, для которого у него нет отладочной информации (т. е. таблицы символов). Функция printf(), будучи скомпонованной из библиотеки C, является примером такого кода.

```

int main(пустота)
{
    целое число i = 4;
    int j = 6;

    printf("i: %d, j: %d\n", i, j);
    поменять местами(i, j);
    printf("i: %d, j: %d\n", i, j);

    возврат 0;
}

недействительный своп(целое а, целое б)
{
    целочисленный с = а;
    а = б;
    б = с;
}

```

Рисунок2-9: *следующий* шаги над функция.

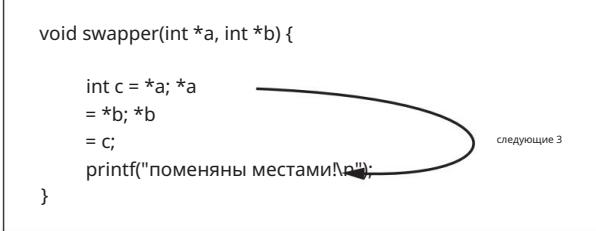
Используете ли вы next или step , на самом деле зависит от того, что вы пытаетесь сделать. делать. Если вы находитесь в части кода, где нет вызовов функций, это не имеет значения какую из них вы используете. В этом случае обе команды полностью эквивалентны.

Однако, если вы отлаживаете программу и обнаруживаете, что собираетесь сделать шаг в функцию, которая, как вы знаете, не содержит ошибок (или не имеет отношения к ошибке, которую вы пытаясь отследить), очевидно, вы хотели бы использовать следующий , чтобы спасти себя от пошаговое выполнение каждой строки функции, которая вас не интересует.

Один из общих принципов отладки, изложенных в Главе 1, заключался в следующем: используйте подход сверху вниз для отладки. Если вы проходите по исходному коду код и сталкиваетесь с вызовом функции, обычно лучше использовать next вместо step. Сразу после использования next в этой ситуации вы бы проверили, был ли результат вызова правильным. Если это так, то ошибка, скорее всего, не в функции, то есть использование next вместо step сэкономило вам время и усилия на stepping через каждую строку функции. С другой стороны, если результат функция неверна, вы можете перезапустить программу, установив временную точку останова при вызове функции, а затем использовать step для входа в функцию.

Обе команды, next и step, принимают необязательный числовой аргумент. который указывает количество дополнительных строк для следующего или шага. В других слова, next 3 то же самое, что набрать next три раза подряд (или набрать next один раз, а затем дважды нажмите клавишу ENTER).⁵ На рисунке 2-10 показана иллюстрация того, что делает next 3 :

⁵ Пользователи Vim должны чувствовать себя как дома, используя концепцию указания количества для заданного команда.

Рисунок2-10: следующий с подсчетом

2.9.1.2 Возобновление выполнения программы с помощью

`continue` Второй метод возобновления выполнения — с помощью команды `continue`, сокращенно `c`. В отличие от `step` и `next`, которые выполняют только одну строку кода, эта команда заставляет GDB возобновлять выполнение вашей программы до тех пор, пока не сработает точка останова или программа не завершится.

Команда `continue` может принимать необязательный целочисленный аргумент `n`. Это число сообщает GDB, что нужно игнорировать следующие `n` точек останова. Например, `continue 3` сообщает GDB, что нужно возобновить выполнение программы и игнорировать следующие 3 точки останова.

2.9.1.3 Возобновление выполнения программы с помощью

`finish` После срабатывания точки останова команды `next` и `step` используются для выполнения программы строка за строкой. Иногда это может быть болезненным занятием.

Например, предположим, что GDB достиг точки останова внутри функции. Вы проверили несколько переменных и собрали всю информацию, которую намеревались получить. На этом этапе вам неинтересно пошаговое выполнение оставшейся части функции. Вы хотели бы вернуться к вызывающей функции, где GDB находился до того, как вы вошли в вызываемую функцию. Однако установка внешней точки останова и использование `continue` кажутся расточительными, если все, что вы хотите сделать, это пропустить оставшуюся часть функции. Вот где вступает в дело `finish`.

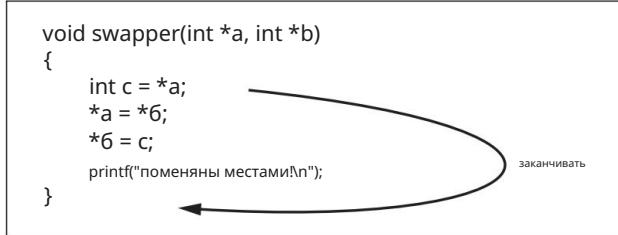
Команда `finish` (сокращенно `fin`) предписывает GDB возобновить выполнение до момента, когда завершится текущий стековый кадр. На английском это означает, что если вы находитесь в функции, отличной от `main()`, команда `finish` заставит GDB возобновить выполнение до момента, когда функция вернет управление. Рисунок 2-11 иллюстрирует использование `finish`.

Хотя вы можете набрать `next 3` вместо `finish`, проще набрать последнее, что подразумевает подсчет строк (все, что больше полудюжины, будет бесполезным неудобством).

Может показаться, что `Finish` не выполняет каждую строку кода, как это происходит в нижней части функции, но это так. GDB выполняет каждую строку без пауз, за исключением случаев, когда нужно показать вывод программы.

Другое распространенное использование `finish` — когда вы случайно вошли в функцию, которую намеревались перешагнуть (другими словами, вы использовали `step`, когда

⁶ Если есть промежуточные точки останова, `Finish` остановится на них.

Рисунок 2-11: ~~заканчивать~~ выполнение текущего возврата функция не возобновит

вы хотели использовать `next`). В этом случае использование `finish` возвращает вас обратно туда, где вы бы были, если бы использовали следующее.

Если вы находитесь внутри рекурсивной функции, `finish` перенесет вас только на один уровень вверх в рекурсии. Это потому, что каждый вызов считается вызовом функции в свое собственное право, поскольку у каждого есть свой собственный стековый фрейм. Если вы хотите полностью выйти из рекурсивной функции, когда рекурсивный уровень высок, временная точка останова вместе с `continue` или использование команды `until` — это более уместно. Мы обсудим до следующего.

2.9.1.4 Возобновление выполнения программы с помощью `until`

Напомним, что команда `Finish` завершает выполнение текущей функции, без дальнейших пауз в функции (за исключением любых промежуточных точек останова). Аналогично, команда `until` (сокращенно просто `u`) обычно используется для завершения цикла выполнения, без дальнейших пауз внутри цикла, за исключением любых промежуточных точек останова внутри цикла. Рассмотрим следующий фрагмент кода.

...предыдущий код...

```

целое число i = 9999;
в то время как (я--) {
    printf("i равно %d\n", i);
    ... МНОГО КОДА ...
}

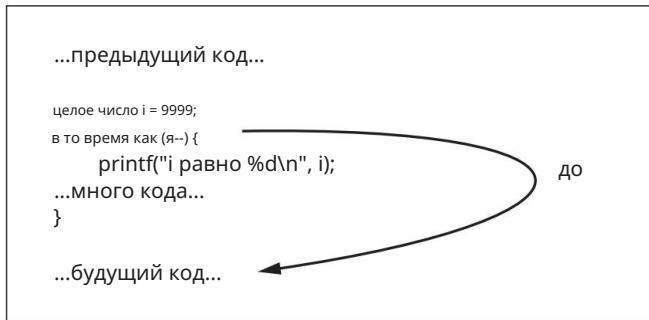
```

...будущий код...

Предположим, что GDB остановлен в точке останова на операторе `while`, вы проверили несколько переменных, и теперь вы хотите выйти из цикла для отладки «будущего кода».

Проблема в том, что `i` настолько велико, что его использование рядом с завершить цикл. Вы не можете использовать команду `finish`, потому что эта команда пройдет прямо над «будущим кодом» и вывести нас из функции. Вы можете установить временную точку останова в будущем коде и использовать `continue`; однако, это именно та ситуация, которую `until` должен был решить.

Использование `until` выполнит оставшуюся часть цикла, оставив GDB на паузе в первую строку кода после цикла. Рисунок 2-12 показывает иллюстрацию того, что использование `until` выполнит:



Фигура 2-12: **ДО** получает следующему нам самая высокая линия ИСТОЧНИК КОД.

Конечно, если GDB обнаружит точку останова до выхода из цикла, он все равно остановится там: Если бы была точка останова в операторе `printf()` в Рисунок 2-12, вы наверняка захотите его отключить.

В руководстве пользователя GDB дано официальное определение `until`:

Выполнять до тех пор, пока программа не достигнет исходной строки, большей, чем текущий [один].

Однако документация также предупреждает, что это может быть немного контрмерой. Интуитивно понятно. Чтобы продемонстрировать почему, рассмотрим следующую программу:

```

#include <stdio.h>

int main(пустота)
{
    целочисленный я;

    для (i=0; i<10; ++i)
        printf("Привет, мир!");

    возврат 0;
}

```

Листинг 2-2: до тех пор, пока-аномалия.c

Мы установим точку останова на входе в `main()`, запустим программу и используем пока не дойдете до заявления о возврате .

```

$ gdb до аномалии
Используется библиотека хоста libthread_db "/lib/tls/libthread_db.so.1".
(gdb) перерыв основной

```

Точка останова 1 по адресу 0x80483b4: файл until-anomaly.c, строка

7. (gdb)

run Запуск программы: until-anomaly

Точка останова 1, main() в until-anomaly.c:7 для (i=0; i<10; ++i)

7 (gdb) до 8

printf("Привет, мир!");

(gdb) до тех пор,

7 пока для (i=0; i<10; ++i) (gdb) до тех

пор, пока не будет

10 возвращено 0;

(гдб)

Ого! Используя until, GDB перешел от строки 7 к строке 8, а затем обратно к строке 7. Это ведь не значит, что исходная строка 7 больше исходной строки 8? На самом деле, значит. Возможно, вы уже догадались, так как это, похоже, общая тема. В конечном итоге GDB работает с машинными инструкциями. Хотя конструкция for написана с проверкой цикла в верхней части тела, GCC скомпилировал программу с условным оператором в нижней части тела цикла. Поскольку условный оператор связан со строкой 7 исходного кода, похоже, что GDB перешел назад в исходном коде. Фактически, until на самом деле выполняет команду до тех пор, пока не достигнет машинной инструкции с более высоким адресом памяти, чем текущий, а не до тех пор, пока не достигнет большего номера строки в исходном коде.

На практике такие вещи могут возникать не так уж часто, но приятно знать, поймите это, когда это происходит. Кроме того, изучая некоторые странности поведения GDB, вы можете почерпнуть информацию о том, как компиляторы преобразуют исходный код в машинные инструкции — неплохие знания.

Если вам любопытно и вы знаете язык ассемблера вашей машины, вы можете быстро просмотреть машинный код, используя команду GDB disassemble , за которой следует p/x \$pc для вывода текущего местоположения.⁷ Это покажет вам, что будет делать until . Но это всего лишь причуда, и на практике это не проблема. Если вы находитесь в конце цикла и выполнение команды until приводит к переходу обратно в начало цикла, просто выполните until второй раз, и вы выйдете из цикла, как и хотели.

Команда until также может принимать аргумент местоположения в исходном коде. Фактически, она принимает те же аргументы, что и команда break , которая обсуждалась в разделе 2.4.1. Возвращаясь к листингу 2-1, если GDB сработала точка останова на входе в main(), все это эквивалентные способы удобного выполнения программы до входа в swap():

- до 17

- до обмена

⁷ Вы также можете просмотреть машинный код, используя опцию -S GCC. Это создаст файл на языке ассемблера с суффиксом .s, показывающий код, созданный компилятором. Обратите внимание, что это создаст только файл на языке ассемблера, а не исполняемый файл.

```
• до swapflaw.c:17 • до
swapflaw.c:swap
```

2.9.2 В DDD

Мы начнем этот раздел с обсуждения различных способов возобновления выполнения после остановки DDD в точке останова.

2.9.2.1 Стандартные операции DDD

имеет кнопки для следующего и следующего шага в командном инструменте. Кроме того, вы можете выполнить шаг и следующий с помощью функциональных клавиш F5 и F6 соответственно.

Если вы хотите использовать next или step с аргументом в DDD, то есть, accomplish что бы вы сделали в GDB через

```
(gdb) следующие 3
```

вам нужно будет использовать сам GDB в окне консоли DDD.

DDD имеет кнопку для продолжения в Command Tool, но опять же, если вы хотите выполнить continue с аргументом, используйте GDB Console. Вы можете щелкнуть левой кнопкой мыши по Source Window, чтобы вызвать опцию «продолжить до этого места», которая на самом деле устанавливает временную точку останова (см. Раздел 2.4.1) на этой строке исходного кода. Точнее, «продолжить до этого места» означает «продолжить до этой точки, но также остановиться на любых промежуточных точках останова».

В GDB можно было бы достичь того же эффекта, что и finish через next , с числовым аргументом, так что finish был бы лишь немного удобнее.

Но в DDD использование Finish — это явный выигрыш, требующий одного щелчка мыши в командной строке.

Если вы используете DDD, вы можете выполнить until , используя кнопку Until на панели команд, или щелкнув левой кнопкой мыши по строке меню Program | Until, или используя сочетание клавиш F7. Из всех этих вариантов вы обязательно найдете один из них удобным! Как и со многими другими командами GDB, если вы хотите использовать until с аргументом, вам нужно будет передать их непосредственно GDB в окне консоли DDD.

2.9.2.2 Отменить/

Повторить Как отмечено в разделе 2.7.6, DDD имеет бесценную функцию Отменить/ Повторить. В этом разделе мы показали, как отменить случайное удаление точки останова. Ее также можно использовать для таких действий, как Выполнить, Далее, Шаг и т. д.

Рассмотрим, например, ситуацию, изображенную на рисунке 2-13. Мы достигли точки останова при вызове swap() и намеревались выполнить операцию Step, но случайно нажали Next. Но нажав Undo, вы можете откатить время назад, как показано на рисунке 2-14. DDD напоминает вам, что вы что-то отменили, отображая свой курсор текущей строки в виде контура вместо сплошного зеленого цвета, который он обычно использует.

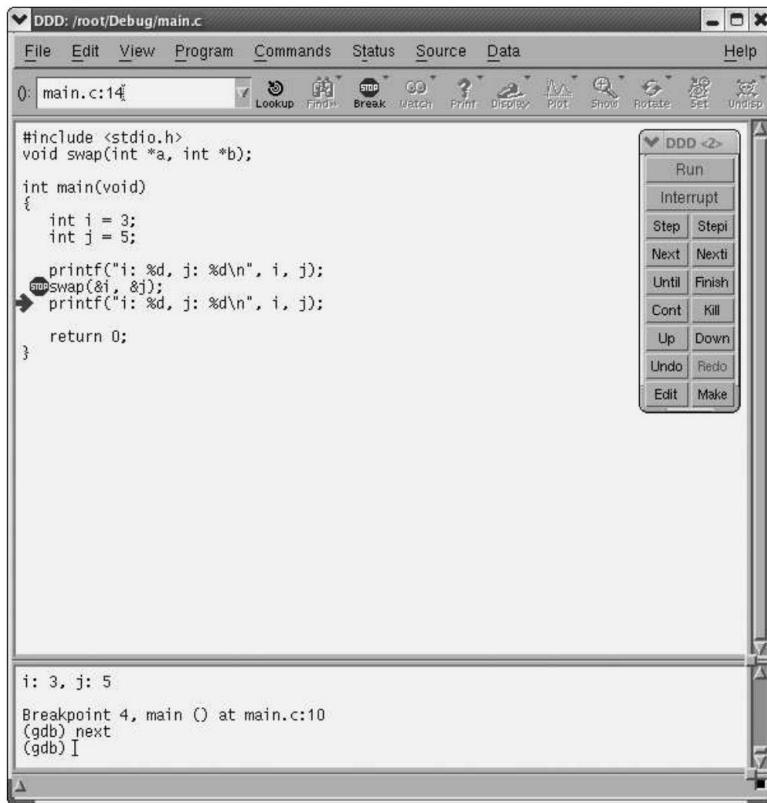


Рисунок 2-13: Упс!

2.9.3 В затмении

Аналогами шагов и следующих в Eclipse являются значки «Шаг внутрь» и «Шаг через».

Значок «Шаг в» виден в представлении «Отладка» (вверху слева) на рисунке 2-15 ;

Указатель мыши временно вызвал появление метки значка. Шаг

Значок Over находится справа от него. Обратите внимание, что следующим оператором, который вы выполните, будет вызов get_args(), поэтому нажатие кнопки Step Into приведет к следующая пауза выполнения происходит на первом операторе внутри этой функции, в то время как выбор Step Over будет означать, что следующая пауза будет при вызове данные_процесса().

В Eclipse есть значок Step Return (рядом с Step Over), который выполняет завершение. Он не имеет ничего, что бы точно соответствовало until. Его Run to Line (вызывается щелкнув целевую строку, а затем щелкнув правой кнопкой мыши в окне исходного кода) как правило, вы достигнете того, чего хотите с помощью until.

2.10 Условные точки останова

Пока точка останова включена, отладчик всегда останавливается на этой точке останова.

Однако иногда полезно указать отладчику остановиться на точке останова-

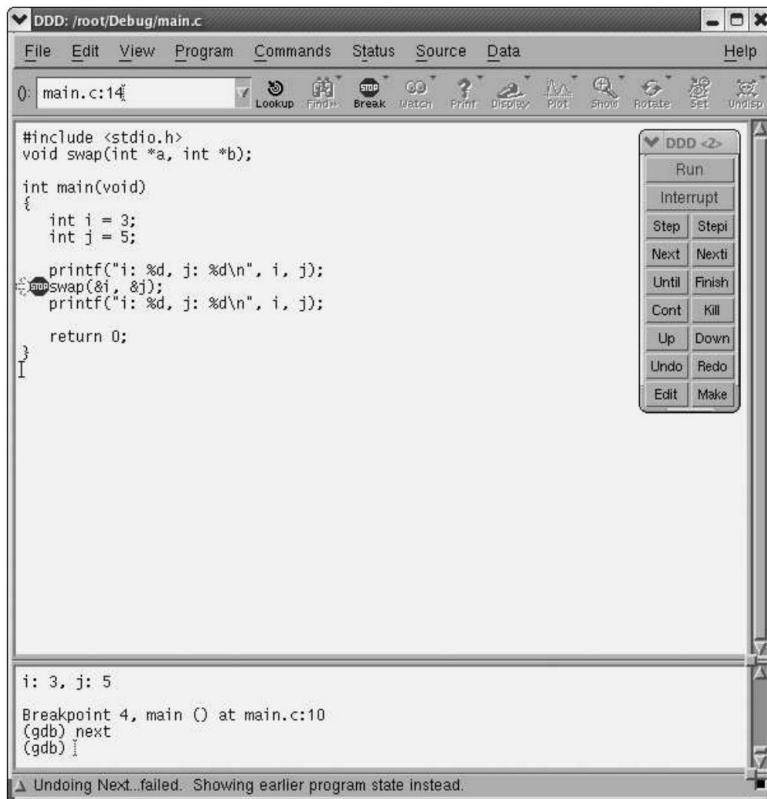


Рисунок2-14: Ввод в зубная паста обратно в тюбик

указывает только в том случае, если выполняется некоторое условие, например, когда переменная имеет особенно интересное значение.

Это похоже на то, как работают контрольные точки, но с важным отличием. Если у вас есть подозрение, где переменная получает фиктивное значение, условная точка останова предпочтительнее точки наблюдения. Точка наблюдения будет break всякий раз, когда эта переменная меняет значение. Условная точка останова сломается только в предполагаемом проблемном коде, и только тогда, когда переменная принимает фиктивное значение. В этом смысле, контрольные точки хороши, когда вы не имеете ни малейшего понятия, где переменная получает свое фиктивное значение. Это особенно полезно для глобальных переменных или локальных переменных, которые постоянно передаются между функциями. Но в большинстве других случаев правильно размещенная условная точка останова более полезна и удобна.

2.10.1 ГБД

Синтаксис установки условной точки останова:

`break break-args если (условие)`

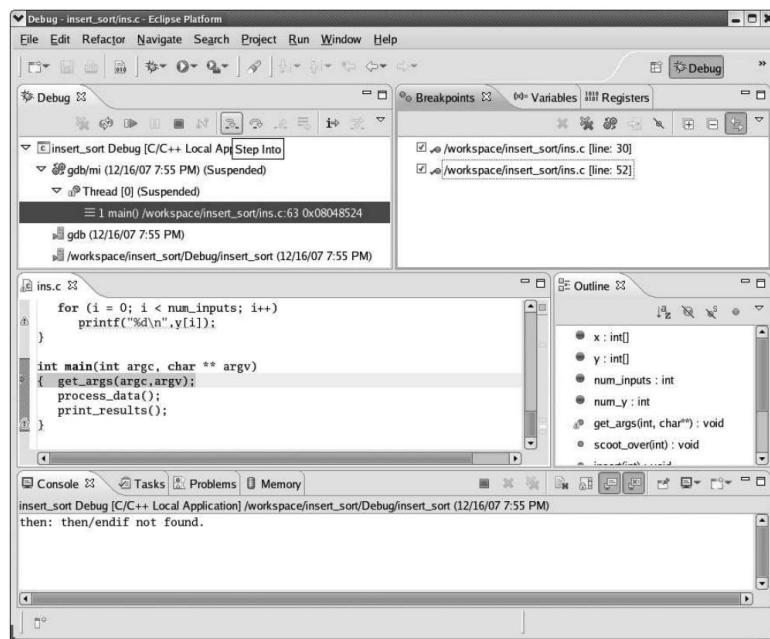


Рисунок 2-15: Шаг в иконку в затмение

где break-args — это любой из аргументов, которые вы можете передать break для указания расположение точки останова, как обсуждалось в разделе 2.4.1, а условие — это булевое выражение, как определено в разделе 2.12.2. Скобки вокруг условия необязательно. Они могут заставить некоторых программистов на С чувствовать себя более дома, но с другой стороны, вы, возможно, предпочтете более лаконичный вид.

Например, вот как бы вы остановились в main(), если бы пользователь ввел некоторые аргументы командной строки программы:⁸

прерывание основного потока, если argc > 1

Условное прерывание чрезвычайно полезно, особенно в циклических конструкциях. В котором что-то плохое происходит при определенном значении индексной переменной. Рассмотрим следующий фрагмент кода:

```
для (я=0; я<=75000; ++я) {
    retval = процесс(и);
    do_something(retval);
}
```

⁸ При условии, что вы объявили argc и argv в качестве аргументов main(). (Конечно, если вы объявили (Имейте их с разными именами, например ac и av, используйте их.) Кстати, обратите внимание, что программа всегда получает как минимум один аргумент — имя самой программы, на которое указывает argv[0], что мы здесь не считаем «аргументом пользователя».

Предположим, вы знаете, что ваша программа выходит из строя, когда *i* равно 70 000. Вы хотите прерваться в верхней части цикла, но не хотите делать это через 69 999 итераций. Вот где условный останов действительно сияет. Вы можете установить точку останова в верхней части цикла, но только когда *i* равно 70 000, с помощью следующего:

```
перерыв, если (i == 70000)
```

Конечно, вы могли бы добиться того же эффекта, набрав, скажем, `continue 69999`, но это было бы менее удобно.

Условное прерывание также чрезвычайно гибко. Вы можете сделать гораздо больше, чем просто проверить переменную на равенство или неравенство. Какие типы вещей вы можете использовать в условии? Практически любое выражение, которое вы можете использовать в допустимом условном операторе С. Все, что вы используете, должно иметь логическое значение, то есть `true` (не ноль) или `false` (ноль). Это включает в себя:

- Операторы равенства, логические операторы и операторы неравенства (`<`, `<=`, `==`, `!=`, `>`, `>=`, `&&`, `||` и т. д.);

например:

```
перерыв 180 если строка==NULL && i < 0
```

- Побитовые и сдвиговые операторы (`&`, `|`, `^`, `>>`, `<<` и т. д.), например:

```
break test.c:34 если (x & y) == 1
```

- Арифметические операторы (`+`, `-`, `*`, `/`, `%`), например:

```
break myfunc если i % (j + 3) != 0
```

- Ваши собственные функции, если они связаны с программой; например:

```
break test.c:myfunc если ! check_variable_sanity(i)
```

- Библиотечные функции, если библиотека связана с вашим кодом, например:

```
перерыв 44 если strlen(mystring) == 0
```

Действуют правила приоритета, поэтому вам может потребоваться использовать скобки вокруг конструкций вроде `(x & y) == 0`.

Кроме того, если вы используете библиотечную функцию в выражении GDB, и библиотека не был скомпилирован с отладочными символами (что почти наверняка так и есть), единственны возвращаемые значения, которые вы можете использовать в условиях точки останова, — это значения типа `int`. Другими словами, без отладочной информации GDB предполагает, что возвращаемое значение функции — это `int`. Если это предположение неверно, возвращаемое значение функции будет неверно истолковано.

```
(gdb) распечатать cos(0.0)
$1 = 14368
```

К сожалению, приведение типов тоже не помогает:

```
(gdb) print (double) cos(0.0) $2 = 14336
```

Если вы плохо разбираетесь в тригонометрии, косинус 0 равен 1.

Использование функций, возвращающих нецелые числа

На самом деле, есть способ использовать функции, которые не возвращают int в выражениях GDB, но он довольно загадочен. Хитрость заключается в определении удобной переменной GDB с соответствующим типом данных, указывающим на функцию.

```
(gdb) set $p = (double (*) (double)) cos (gdb) ptype $p
type = double (*())
(gdb) p cos(3.14159265)
$2 = 14368 (gdb) p
$p(3.14159265)
$4 = -1
```

Если вы все еще не знаете тригонометрию, 3,14159265 приблизительно равно π, а косинус π равен -1.

Можно задать условия для обычных точек останова, чтобы превратить их в условные точки останова. Например, если вы установили точку останова 3 как безусловную, но теперь хотите добавить условие i == 3, просто введите

```
(gdb) условие 3 i == 3
```

Если позже вы захотите удалить условие, но сохранить точку останова, просто введите

```
(gdb) условие 3
```

2.10.2 DDD

Вы можете установить условные точки останова с помощью DDD, используя семантику GDB с окном консоли. Или используйте DDD следующим образом. Установите обычную (т. е. безусловную) точку останова в том месте кода, где вы хотите установить условную точку останова. Щелкните правой кнопкой мыши и удерживайте красный знак остановки, чтобы открыть меню, и выберите Свойства. Появится всплывающее окно с полем ввода текста с надписью Условие. Введите условие в этом поле, щелкните Применить, а затем щелкните Закрыть. Теперь точка останова является условной.

Это показано на рисунке 2-16. Мы видим условие $j == 0$ на точке останова 4. Кстати, знак остановки на этой строке теперь будет содержать вопрос знак, чтобы напомнить нам, что это условный перерыв.

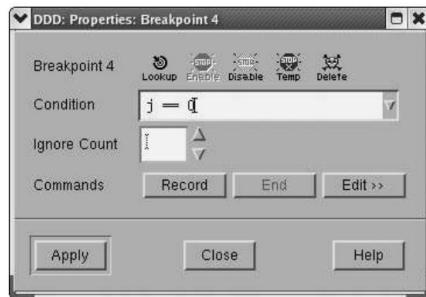


Рисунок2-16: а состояния на точка останова в ДДД

2.10.3 Затмение

Чтобы сделать точку останова условной, щелкните правой кнопкой мыши по символу точки останова для этой точки. выберите Свойства точки останова... | Общие и заполните условие в диалоговое окно. Диалоговое окно изображено на рисунке 2-17.

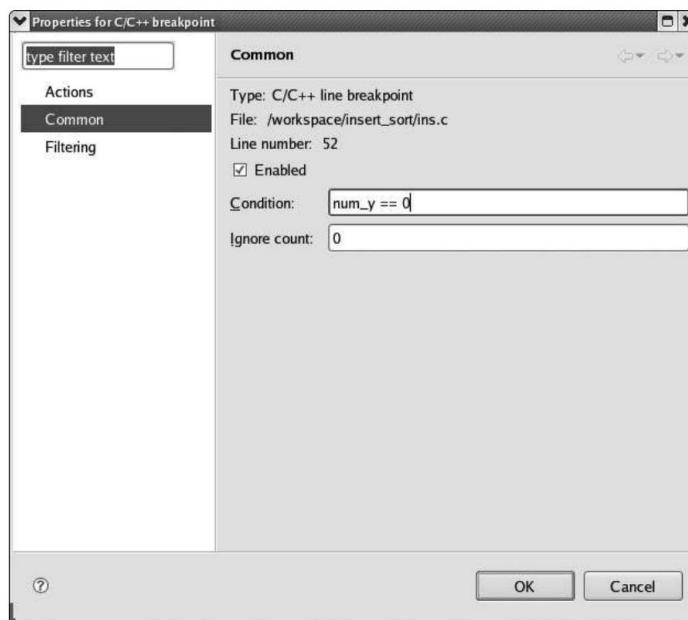


Рисунок2-17: а состояния на точка остановав Затмение

2.11 Списки команд точек останова

После того, как GDB достигает точки останова, вы почти всегда будете проверять переменную. Если одна и та же точка останова срабатывает повторно (как в случае с точкой останова внутри цикла), вы будете проверять одну и ту же переменную повторно. Разве не было бы неплохо автоматизировать процедуру, указав GDB автоматически выполнять набор команд каждый раз, когда он достигает точки останова?

На самом деле, вы можете сделать это с помощью «списков команд точек останова». Мы будем использовать Команду GDB `printf` для иллюстрации списков команд. Вы еще не были формально с ней знакомы, но `printf` в основном работает в GDB так же, как и в C, но скобки необязательны.

Списки команд задаются с помощью команды `commands`:

```
команды номер точки останова
...
команды
...
конец
```

где `breakpoint-number` — это идентификатор точки останова, к которой вы хотите добавить команды, а `commands` — это разделенный новой строкой список любых допустимых команд GDB. Вы вводите команды одну за другой, а затем набираете `end`, чтобы обозначить, что вы закончили ввод команд. После этого, когда бы GDB ни останавливался на этой точке останова, он будет выполнять все команды, которые вы ему дали. Давайте рассмотрим пример.

Рассмотрим следующую программу:

```
#include <stdio.h> int
fibonacci(int n);

целочисленный основной(void)
{
    printf("Фибоначчи(%d) равен %d.\n", fibonacci(3));
    возврат 0;
}

целое число фибоначчи(целое число
n) {
    если ( n <= 0 || n == 1 )
        вернуть 1;
    еще
        вернуть фибоначчи(n-1) + фибоначчи(n-2);
}
```

Листинг 2-3: `fibonacci.c`

Мы хотели бы увидеть, какие значения передаются в `fibonacci()` и в каком порядке. Однако вам не нужно вставлять операторы `printf()` и перекомпилировать

code. Во-первых, это было бы неуклюже в книге по отладке, не так ли? Но что еще важнее, потребуется время, чтобы вставить код и перекомпилировать/связать, а затем удалить этот код и перекомпилировать/связать после исправления этой конкретной ошибки, особенно если ваша программа большая. Более того, это загромождит ваш код операторами, не связанными с кодом, что затруднит его чтение в процессе отладки.

Вы можете пройти по коду и вывести `n` при каждом вызове `fibonacci()`, но списки команд лучше, потому что они устраниют необходимость многократно вводить команду печати. Давайте посмотрим.

Сначала установите точку останова в верхней части `fibonacci()`. Эта точка останова будет присвоен идентификатор 1, так как это первая установленная вами точка останова. Затем установите команду на точке останова 1, чтобы напечатать переменную `n`.

```
$ gdb fibonacci (gdb)
break fibonacci Точка
останова 1 по адресу 0x80483e0: файл fibonacci.c, строка 13. (gdb)
commands 1 Введите
команды для случай достижения точки останова 1, по одной на строку.
Завершите строкой, в которой просто
написано "end". >printf "fibonacci was passed %d.\n",
n >end
(gdb)
```

Теперь запустите программу и посмотрите, что произойдет.

```
(gdb) запустить
Стартовая программа: Фибоначчи
```

```
Точка останова 1, Фибоначчи (n=3) в fibonacci.c:13 если ( n <= 0 || n
== 1 ) 13 Фибоначчи был пройден 3.
(gdb) continue Продолжение.
```

```
Точка останова 1, Фибоначчи (n=2) в fibonacci.c:13 если ( n <= 0
13           || n == 1 ) Фибоначчи был
пройден 2. (gdb) continue
Продолжение.
```

```
Точка останова 1, Фибоначчи (n=1) в fibonacci.c:13 если ( n <= 0
13           || n == 1 ) Фибоначчи был
пройден 1. (gdb) continue
Продолжение.
```

```
Точка останова 1, Фибоначчи (n=0) в fibonacci.c:13 если ( n <= 0 || n
== 1 ) 13
```

Фибоначчи был передан как
0. (gdb) продолжить
Продолжаем.

Точка останова 1, Фибоначчи (n=1) в fibonacci.c:13 если (n <= 13
0 || n == 1) Фибоначчи был пройден 1. (gdb) continue
Продолжение.

Фибоначчи(3) равно 3.

Программа завершилась нормально.

(gdb)

Ну, это почти то, чего мы ожидали, но вывод слишком многословен. В конце концов, мы уже знаем, где находится точка останова. К счастью, вы можете заставить GDB меньше беспокоиться о срабатывании точек останова с помощью команды silent , которая должна быть первым элементом в списке команд. Давайте посмотрим на silent в действии. Обратите внимание, как мы переопределяем список команд, помещая новый список команд «поверх» того, который мы установили ранее:

(gdb) команды 1 Введите команды, которые будут выполняться при достижении точки останова 1, по одной в каждой строке.
Завершить фразой «конец». >без звука

>printf "Фибоначчи был передан %d.\n", n >end

(гдб)

И вот что получилось:

(gdb) run
Запуск программы: fibonacci
fibonacci передано 3. (gdb)
continue
Продолжение.
fibonacci передано 2. (gdb)
continue
Продолжение.
fibonacci передано 1. (gdb)
continue
Продолжение.
fibonacci передано 0. (gdb)
continue
Продолжение.
fibonacci передано 1. (gdb)
continue
Продолжение.

Фибоначчи(3) равно 3.

Программа завершилась нормально.

(gdb)

Отлично. Последняя возможность для демонстрации: если последняя команда в списке команд — continue, GDB автоматически продолжит выполнение программы после того, как она завершит команды в списке команд:

(gdb) команда 1 Введите

команды, которые будут выполняться при достижении точки останова 1, по одной в каждой строке.

Завершите строкой, в которой просто написано

```
"end". >silent >printf "fibonacci was passed %d.\n", n  
>continue
```

>конец

(gdb)

запустить Запуск программы: fibonacci

fibonacci передано 3. fibonacci

передано 2. fibonacci передано 1.

fibonacci передано 0. fibonacci

передано 1.

Фибоначчи(3) равно 3.

Программа завершилась нормально.

(gdb)

Возможно, вы захотите сделать что-то подобное в других программах или в других местах. строки этой программы, поэтому давайте сделаем из нее макрос, используя команду GDB define .

Сначала давайте определим макрос, который мы назовем print_and_go:

(gdb) define print_and_go

Переопределить команду "print_and_go"? (у или н) у Введите
команды для определения "print_and_go".

Завершите строкой, в которой просто

```
написано «end». >printf  
$arg0, $arg1 >continue
```

>конец

Чтобы использовать его, как указано выше, введите:

(gdb) команда 1 Введите

команды, которые будут выполняться при достижении точки останова 1, по одной в каждой строке.

Завершить фразой «конец», >без звука

```
>print_and_go "fibonacci() был передан %d\n" n >end
```

Обратите внимание, что между аргументами print_and_go нет запятой . После этого вы получите тот же вывод, что и раньше, когда запустите программу, но суть в том, что теперь вы можете использовать его в общем, в любом месте кода. Более того, вы можете поместить его в свой файл .gdbinit для использования в других программах. Кстати, допускается до десяти аргументов, хотя в этом примере их всего два.

Вы можете получить список всех макросов, введя команду show user.

Списки команд очень полезны, но вы также можете комбинировать их с условным прерыванием, и это мощно. С таким условным вводом/выводом у вас может даже возникнуть соблазн выбросить C и просто использовать GDB в качестве языка программирования по вашему выбору. Шутка, конечно.

Команды и выражения

На самом деле, любое допустимое выражение GDB, как описано в разделе 2.12.2, может быть включено в список команд. Вы можете использовать библиотечные функции или даже свои собственные функции, если они связаны с исполняемым файлом, и вы можете использовать их возвращаемое значение, если они возвращают int. Например:

```
(gdb) команда 2 Введите
команды, которые будут выполняться при достижении точки останова 2, по одной в каждой строке.
Завершите строкой, в которой написано
просто
"end". >silent >printf "string has length of %d\n", strlen(string) >end
```

Списки команд в DDD аналогичны условным точкам останова в DDD.

Сначала установите точку останова. Щелкните правой кнопкой мыши по красному значку остановки и выберите «Свойства». Появится всплывающее окно. Большое подокно будет справа (если вы не видите большое подокно, щелкните левой кнопкой мыши кнопку «Изменить», которая переключает видимость окна команд). Вы можете вводить свои команды прямо в этом окне. Также есть кнопка «Запись». Если вы щелкните правой кнопкой мыши по этой кнопке, вы можете вводить свои команды в консоль GDB.

Судя по всему, в Eclipse нет функции списка команд.

2.12 Точки наблюдения

Точка наблюдения — это особый вид точки останова, которая, как и обычная точка останова, является инструкцией, которая сообщает GDB о необходимости приостановить выполнение вашей программы. Разница в том, что точки наблюдения не «живут» в строке исходного кода. Вместо этого точка наблюдения — это инструкция, которая сообщает GDB о необходимости приостановить выполнение всякий раз, когда

выражение изменяет значение.⁹ Это выражение может быть довольно простым, как имя переменной:

(gdb) смотреть я

что заставит GDB остановиться всякий раз, когда *i* изменяет значение. Выражение может также может быть довольно сложным:

(gdb) смотреть (*i* | *j* > 12) && *i* > 24 && strlen(имя) > 6

Вы можете представить себе точку наблюдения как «прикрепленную» к выражению; когда значение этого выражения изменится, GDB приостановит выполнение программы исполнение.

Хотя контрольные точки и точки останова управляются одинаково, есть является важным различием между ними. Точка останова связана с расположение в исходном коде. Поскольку ваш код не меняется, есть нет риска, что строка кода «выйдет за рамки». Поскольку в языке С жесткая область действия rules, вы можете наблюдать только за переменной, которая существует и находится в области видимости. Как только переменная больше не существует ни в одном кадре стека вызовов (когда функция, содержащая локальную переменную, возвращается), GDB автоматически удаляет точку наблюдения.

Например, в разделе 2.5 мы рассмотрели программу под названием swap. эта программа, локальная переменная с использовалась для временного хранения. Вы бы не сможет установить наблюдение за *c*, пока GDB не достигнет строки 3 swapper.c, где с определен. Кроме того, если вы установили наблюдение за *c*, наблюдение будет автоматически удаляется после возврата GDB из swapper(), поскольку *c* не будет больше не будет существовать.

Остановка GDB при каждом изменении переменной может быть чем-то вроде неприятность в плотно завернутых циклах или повторяющемся коде. Хотя watch-points звучат отлично, хорошо размещенные breakpoints могут быть гораздо более полезными. Однако watch-points бесценны, если одна из ваших переменных изменяется, особенно глобальная переменная, и вы понятия не имеете, где и как она изменилась. Если вы при работе с многопоточным кодом точки наблюдения имеют ограниченную полезность; GDB способен только наблюдать за переменной в пределах одного потока. Вы можете прочитать Подробнее об этом в Главе 5.

2.12.1 Параметр Точки наблюдения

Когда переменная *var* существует и находится в области видимости, вы можете установить точку наблюдения это с помощью команды

смотреть var

что приведет к остановке GDB при каждом изменении значения переменной *var*.

Вот как многие думают о точках наблюдения, потому что это просто и удобно; однако, есть еще кое-что. Что GDB на самом деле делает, так это break если местоположение памяти для *var* меняет значение. Обычно это не имеет значения

⁹ Выражения будут более подробно рассмотрены в разделе 2.12.2.

Аппаратные точки наблюдения?

При установке точки наблюдения вы можете увидеть сообщение об «аппаратной» точке наблюдения:

```
(gdb) смотреть я  
Аппаратная точка наблюдения 2: i
```

или перечислите свои контрольные точки:

```
(gdb) информация о контрольных точках  
Num Type 2 Disp Enb Адресс Что я  
аппаратная точка наблюдения держать у
```

Очевидно, что 2 относится к идентификатору точки наблюдения, а *i* — это отслеживаемая переменная, но что это за «аппаратная» штука?

На многих платформах есть выделенное оборудование, которое можно использовать для реализации точки наблюдения. GDB попытается использовать оборудование, если оно доступно, поскольку реализация чего-либо с использованием оборудования выполняется быстро.

Возможно, GDB не может установить аппаратные точки наблюдения (платформа может не иметь необходимого оборудования или оборудование может быть занято чем-то другим). Если это так, GDB попытается реализовать точку наблюдения с помощью методов VM, что также быстро.

Если ни один из методов недоступен, GDB реализует сама точка наблюдения, через программное обеспечение, которое может быть очень, очень медленным.

Ваш ЦП может реализовать только ограниченное количество аппаратно-поддерживаемых остановок, включая контрольные точки и аппаратно-поддерживаемые остановочные точки. Это число зависит от архитектуры, но если вы превысите этот лимит, GDB выведет сообщение:

Остановлено; невозможно вставить точки останова.

Возможно, вы запросили слишком много аппаратных точек останова и наблюдения.

Если вы видите это сообщение, вам следует удалить или отключить некоторые из ваших точек наблюдения или аппаратных точек останова.

независимо от того, думаете ли вы о точке наблюдения как о наблюдении за переменной или адресом переменная, но она может быть важна в особых случаях, например, при работе с указатели на указатели.

Давайте рассмотрим один пример сценария, в котором точки наблюдения были бы очень важны. Полезно. Предположим, у вас есть две переменные типа `int`, `x` и `y`, и где-то в код, который вы выполняете `r = &y`, когда вы хотели сделать `r = &x`. Это может привести к утаинственно меняет значение где-то в коде. Фактическое местоположение возникающая ошибка может быть хорошо скрыта, поэтому точка останова может быть не очень полезно. Однако, установив точку наблюдения, вы можете мгновенно узнать, когда и где у изменяет значение.

Это еще не все. Вы не ограничены наблюдением за переменной. Фактически, вы можете наблюдать за выражением, включающим переменные. Всякий раз, когда выражение изменяет значение, GDB сломается. В качестве примера рассмотрим следующий код:

```

1 #include <stdio.h>
2 целые i = 0;
3
4 целочисленный основной(пустой)
5 {
6     я = 3;
7     printf("i равно %d.\n", i);
8
9     я = 5;
10    printf("i равно %d.\n", i);
11
12    возврат 0;
13 }
```

Мы хотели бы знать, когда `i` становится больше 4. Давайте поставим точка останова на входе в `main()`, чтобы получить `i` в области видимости, и установить точку наблюдения, чтобы сообщить вам, когда `i` станет больше 4. Вы не можете установить точку наблюдения на `i`, потому что до запуска программы `i` не существует. Поэтому вам нужно установить сначала установите точку останова на `main()`, а затем точку наблюдения на `i`:

```

(gdb) перерыв основной
Точка останова 1 по адресу 0x80483b4: файл test2.c, строка 6.
(gdb) запустить
Стартовая программа: test2
```

Точка останова 1, `main()` в `test2.c:6`

Теперь, когда `i` находится в области видимости, установите точку наблюдения и сообщите GDB о необходимости продолжить выполнение. Сокращение программы. Мы полностью ожидаем, что `i>4` станет истинным в строке 9.

```

1 (gdb) watch i > 4 2
Аппаратная точка наблюдения 2: i > 4 3
(gdb) continue 4
Продолжение.
5 Аппаратная точка наблюдения 2: i > 4
6
7 Старое значение = 0
8 Новое значение = 1
9 main() в test2.c:10

```

Конечно же, GDB останавливается между строками 9 и 10, где выражение `i>4` изменило значение с 0 (не верно) на 1 (верно).

Вы можете использовать этот метод для установки точек наблюдения с помощью DDD в окне консоли; однако, возможно, вам будет удобнее использовать интерфейс GUI. В окне исходного кода найдите переменную, для которой вы хотите установить точку наблюдения. Это не обязательно должен быть первый экземпляр переменной; вы можете использовать любое упоминание переменной в исходном коде. Щелкните левой кнопкой мыши переменную, чтобы выделить ее, затем щелкните левой кнопкой мыши символ точки наблюдения в строке меню значка. Не будет индикации того, что установлена точка наблюдения, как красный знак остановки для точек останова. Чтобы увидеть, какие точки наблюдения вы установили, вам придется фактически перечислить все ваши точки останова, что мы рассмотрели в разделе 2.3.2.

В Eclipse можно установить точку наблюдения, щелкнув правой кнопкой мыши в исходном окне, выбрав «Добавить выражение наблюдения», а затем заполнив нужное выражение в диалоговом окне.

2.12.2 Выражения

Мы видели пример использования выражения с командой GDB `watch`. Оказывается, есть довольно много команд GDB, например `print`, которые также принимают аргументы выражения. Поэтому, вероятно, нам следует упомянуть о них немного больше.

Выражение в GDB может содержать много вещей:

- Удобные переменные GDB
- Любая переменная в области действия вашей программы, например `i` из предыдущего примера • Любая строковая, числовая или символьная константа • Макросы препроцессора, если программа была скомпилирована с включением информации об отладке препроцессора¹⁰
- Условные операторы, вызовы функций, приведения типов и операторы, определенные используемым вами языком

¹⁰ На момент написания этой статьи официальное руководство пользователя GNU GDB утверждает, что макросы препроцессора не могут использоваться в выражениях; однако это не так. Если вы скомпилируете программу с опцией `GCC -f宏`, макросы препроцессора могут использоваться в выражениях.

Итак, если вы отлаживаете программу на Fortran-77 и хотите узнать, когда переменная *i* стала больше 4, вместо использования `watch i>4`, как вы делали в предыдущем разделе, вы должны использовать `watch i .GT. 4`.

Вы часто видите руководства и документацию, в которых для выражений GDB используется синтаксис C, но это связано с повсеместной распространностью C и C++; если вы используете язык, отличный от C, выражения GDB строятся из элементов этого языка.

3

ПРОВЕРКА И НАСТРОЙКА ПЕРЕМЕННЫЕ



В Главе 1 вы узнали , что в GDB можно распечатать значение переменной с помощью команды print, и что это можно сделать в DDD и Eclipse, переместив указатель мыши на экземпляр переменной в любом месте исходного кода. Но и GDB, и GUI также предлагают гораздо более мощные способы проверки переменных и структур данных, как мы увидим в этой главе.

3.1 Наш основной пример кода

Ниже приведена простая (хотя не обязательно эффективная, модульная и т. д.) реализация двоичного дерева:

```
// bintree.c: процедуры для вставки и сортированной печати двоичного дерева
```

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

```

struct node { int
    val; // сохраненное значение struct node *left; //
    указатель на меньший дочерний элемент struct node
    *right; // указатель на больший дочерний элемент
};

typedef структурный узел *nsp;

корень nsp;

nsp makenode(int x {

    nsp tmp;

    tmp = (nsp) malloc(sizeof(struct node)); tmp->val = x;
    tmp->left = tmp-
    >right = 0; return tmp;

}

недействительная вставка(nsp *btp, int x)
{
    nsp tmp = *бтп;

    если (*бтп == 0) { *бтп
        = makenode(x); return;

    }

    в то время как
    (!) {
        если (x < tmp->val) {

            если (tmp->left != 0) { tmp =
                tmp->left; } иначе
                { tmp->left
                    = makenode(x); break;

            }

        } еще {

            если (tmp->right != 0) { tmp =
                tmp->right; } иначе
                { tmp-
                    >right = makenode(x); break;

            }

        }

    }

}

```

```

        }

    }

}

void printtree(nsp bt) {

    если (bt == 0) return;
    printtree(bt->left);
    printf("%d\n",bt->val);
    printtree(bt->right);
}

int main(int argc, char *argv[]) { int i;

корень =
0; для (i = 1; i < argc; i++)
    вставка(&корень, atoi(argv[i]));
printtree(корень);
}

```

В каждом узле все элементы левого поддерева меньше значения в данном узле, а все элементы правого поддерева больше или равны значению в данном узле. Функция `insert()` создает новый узел и помещает его в соответствующее положение в дереве. Функция `printtree()` отображает элементы любого поддерева в порядке возрастания номеров, в то время как `main()` запускает тест, выводя весь отсортированный массив.¹

Для примеров отладки предположим, что вы случайно закодировал второй вызов `makenode()` в `insert()` как

```
tmp->left = makenode(x);
```

вместо

```
tmp->right = makenode(x);
```

Если запустить этот ошибочный код, что-то сразу пойдет не так:

```
$ бинтри 12 8 5 19 16
16
12
```

¹ Кстати, обратите внимание на `typedef` в строке 12, `nsp`. Это означает node struct pointer, но наш издатель думает, что это No Starch Press.

Давайте рассмотрим, как различные команды проверки в инструментах отладки могут помочь ускорить поиск ошибки.

3.2 Расширенная проверка и настройка переменных

Наш пример дерева здесь имеет большую сложность, и поэтому нужны более сложные методы. Мы рассмотрим некоторые из них здесь.

3.2.1 Инспекция в ГБД

В предыдущих главах вы использовали базовую команду печати GDB . Как вы можете использовать ее здесь? Ну, основная работа, очевидно, выполняется в insert(), так что это было бы хорошим местом для начала. При запуске GDB в цикле while в этой функции вы можете выдавать набор из трех команд GDB каждый раз, когда вы достигаете точки останова:

```
(gdb) p tmp->val $1
= 12
(gdb) p tmp->left $2
= (узел структуры *) 0x8049698 (gdb)
p tmp->right $3 = (узел
структурой *) 0x0
```

(Вспомним из Главы 1 , что выходные данные GDB помечены как \$1, \$2 и и так далее, с этими величинами, которые в совокупности называются историей значений. Мы обсудим их далее в разделе 3.4.1.)

Здесь вы обнаружите, что узел, на который в данный момент указывает tmp , содержит 12 с ненулевым левым указателем, но нулевым правым указателем. Конечно, фактическое значение левого указателя, то есть фактический адрес памяти, вероятно, не представляет здесь прямого интереса, но тот факт, что указатель ненулевой или нулевой, важен. Дело в том, что вы видите, что в данный момент есть левое поддерево ниже 12, но нет правого поддерева.

Первое улучшение: распечатайте структуру целиком

Было бы довольно утомительно продолжать вводить эти три команды печати каждый раз, когда мы достигаем точки останова. Вот как мы могли бы сделать то же самое с помощью всего одной команды печати :

```
(gdb) p *tmp
$4 = {val = 12, left = 0x8049698, right = 0x0}
```

Поскольку tmp указывает на структуру, то *tmp — это сама структура, и, таким образом, GDB показывает нам все содержимое.

Второе улучшение: используйте команду GDB display . Ввод p *tmp

выше экономит времени и усилия. Каждый раз, когда вы достигаете точки останова, вам нужно будет вводить только одну команду GDB, а не три. Но если вы знаете, что будете вводить ее каждый раз, когда достигнете точки останова, вы можете сэкономить еще больше времени и усилий, используя команду GDB display , сокращенно

ated disp. Эта команда сообщает GDB, что нужно автоматически печатать указанный элемент каждый раз, когда возникает пауза в выполнении (из-за точки останова, команд next или step и т. д.):

```
(gdb) disp *tmp 1:
*tmp = {val = 12, left = 0x8049698, right = 0x0} (gdb) Продолжение.
```

```
Точка останова 1, вставить (btp=0x804967c, x=5) в bintree.c:37 if (x < tmp->val)
37           { 1: *tmp = {val = 8, left =
0x0, right = 0x0}
```

Как видно здесь, GDB автоматически вывел *tmp после достижения точки останова, поскольку вы ввели команду display .

Конечно, переменная в списке отображения будет отображаться только в то время, когда она находится в области действия.

Третье улучшение: используйте команду GDB commands Предположим, вы хотите посмотреть значения в дочерних узлах, когда находитесь в данном узле. Вспоминая команду GDB commands из Главы 1, вы можете сделать что-то вроде этого:

```
(gdb) b 37
Точка останова 1 по адресу 0x8048403: файл bintree.c, строка 37.
(gdb) commands 1
Введите команды для случая достижения точки останова 1, по одной на строку.
Завершите строкой, в которой просто
написано
"end". >p tmp->val >if (tmp-
>left != 0) >p tmp->left-
>val
>else >printf "%s\n", "none"
>end
>if (tmp->right != 0) >p tmp-
>right->val >else >printf
"%s\n",
"none" >end

>конец
```

Обратите внимание, что в этом примере команда печати GDB имеет более мощного аналога — printf(), форматирование которого похоже на форматирование ее тезки на языке C.

Вот пример результирующего сеанса GDB:

```
Точка останова 1, вставить (btp=0x804967c, x=8) в bintree.c:37 if (x < tmp->val)
```

37

7 долларов = 12

```
НИКТО
НИКТО
(gdb) с
Продолжение.
```

```
Точка останова 1, вставить (btp=0x804967c, x=5) в bintree.c:37 if (x < tmp->val)
37
6 долларов = 12
7 долларов = 8
НИКТО
(gdb) с
Продолжение.
```

```
Точка останова 1, вставить (btp=0x804967c, x=5) в bintree.c:37 if (x < tmp->val)
37
$8 = 8
НИКТО
НИКТО
```

Конечно, вы можете сделать вывод более нарядным, добавив этикетки и т. д.

Четвертое улучшение: используйте команду вызова GDB. Обычный подход при отладке — изолировать первый элемент данных, в котором возникает проблема. В данном контексте это можно сделать, распечатав все дерево каждый раз, когда вы завершаете вызов `insert()`.

Поскольку в вашем исходном файле в любом случае есть функция, которая делает это — `printtree()` — вы можете просто добавить вызов этой функции сразу после вызова `insert()` в исходном коде:

```
для (i = 1; i < argc; i++) {
    вставить(&корень,atoi(argv[i])); printtree
    (корень);
}
```

Однако это было бы нежелательно с разных точек зрения. Это означало бы, например, что вам пришлось бы тратить время на редактирование исходного файла и его перекомпиляцию. Первое отвлекало бы, а второе могло бы занять некоторое время, если бы у вас была большая программа. В конце концов, именно этого вы и пытаетесь избежать, используя отладчик.

Вместо этого было бы неплохо сделать то же самое из GDB. Вы можете сделать это с помощью команды GDB `call`. Например, вы можете установить точку останова на строке 57, в конце `insert()`, а затем сделать следующее:

```
(gdb) команды 2 Введите
команды, которые будут действовать при достижении точки останова 1, по одной на строку.
Завершите строкой, в которой просто
написано "end". >printf "***** текущее дерево
*****" >call
printtree(root)>end
```

Пример полученного сеанса GDB:

Точка останова 2, вставить (btp=0x8049688, x=12) в bintree.c:57 57 } ***** текущее

дерево *****

12

(gdb) c

Продолжение.

Точка останова 2, вставить (btp=0x8049688, x=8) в bintree.c:57 57 }

***** текущее дерево *****

8

12

(gdb) c

Продолжение.

Точка останова 2, вставить (btp=0x8049688, x=5) в bintree.c:57 57 }

***** текущее дерево *****

5

8

12

(gdb) c

Продолжение.

Точка останова 2, вставить (btp=0x8049688, x=19) в bintree.c:57 57 } *****

текущее дерево *****

19

12

Обратите внимание, что это показывает, что первым элементом данных, вызвавшим проблемы, был номер 19. Эта информация позволит вам очень быстро нацелиться на ошибку. Вы бы перезапустили программу с теми же данными, установив точку останова в начале insert() , но с условием x == 19, а затем исследовали бы, что там происходит.

Набор команд для данной точки останова можно изменять динамически или просто отменять, переопределяя пустой набор:

(gdb) команды 1 Введите

команды, которые будут выполняться при достижении точки останова 1, по одной в каждой строке.

Завершите строкой «end». >end

3.2.2 Инспекция^в DDD

Как вы уже знаете, вы можете вызвать любую команду GDB из DDD в окне консоли DDD. Но одно из реальных преимуществ DDD заключается в том, что многие команды GDB могут быть более удобно запущены в DDD, а в некоторых случаях DDD способен выполнять мощные операции, не предлагаемые GDB, такие как отображение связанных структур данных, описанных ниже.

Как упоминалось в предыдущих главах, проверка значения переменной в DDD очень удобна: просто переместите указатель мыши на любой экземпляр переменной в окне исходного кода. Но есть и другие полезные штуки, которые стоит использовать, особенно для программ, использующих связанные структуры данных. Мы проиллюстрируем это на примере того же двоичного дерева, что и ранее в этой главе.

Вспомним иллюстрацию команды GDB `display` :

```
(gdb) disp *tmp
```

Здесь содержимое структуры, на которую указывает `tmp`, автоматически выводится на печать каждый раз, когда вы достигаете точки останова или иным образом останавливаете выполнение программы. Поскольку `tmp` был указателем на текущий узел в дереве, эта автоматическая печать была полезна для отслеживания вашего прогресса в обходе дерева.

Несколько аналогична команда `Display` в DDD. Если щелкнуть правой кнопкой мыши по любому экземпляру переменной, скажем, `root` в этом примере, в окне исходного кода, появится меню, как показано на рисунке 3-1. Как вы можете видеть, у вас есть несколько вариантов выбора для просмотра `root`. Варианты `Print root` и `Print *root` работают точно так же, как их аналоги GDB, и фактически их вывод отображается в консоли DDD (где команды GDB отображаются/вводятся). Но для рассматриваемого здесь случая наиболее интересным вариантом является `Display *root`. Результат выбора этого варианта после достижения точки останова на строке 48 исходного кода показан на рисунке 3-2.

Появилось новое окно DDD — Окно данных с узлом, соответствующим корню. Пока что это не более чем графический аналог команды GDB `display`. Но что здесь действительно приятно, так это то, что вы можете следовать за ссылками дерева! Например, чтобы следовать за левой ветвью дерева, щелкните правой кнопкой мыши левое поле отображаемого корневого узла. (Вы не будете делать этого в правом узле в данный момент, так как ссылка равна 0.) Затем выберите опцию `Display (*)` во всплывающем меню, и теперь DDD выглядит так, как показано на рисунке 3-3. Итак, DDD представляет вам рисунок дерева (или этой его части), как будто вы пишете сами на доске — очень круто!

Отображение содержимого существующего узла будет автоматически обновляться всякий раз, когда содержимое узла изменяется. Каждый раз, когда ссылка меняется с нуля на ненулевое значение, вы можете щелкнуть правой кнопкой мыши по ней, чтобы отобразить новый узел.

Очевидно, что окно данных может быстро засориться. Вы можете расширить окно, щелкнув и перетащив маленький квадрат в правом нижнем углу этого окна, но лучшим подходом было бы предвидеть эту ситуацию еще до запуска DDD. Если вы вызываете DDD с отдельной опцией командной строки

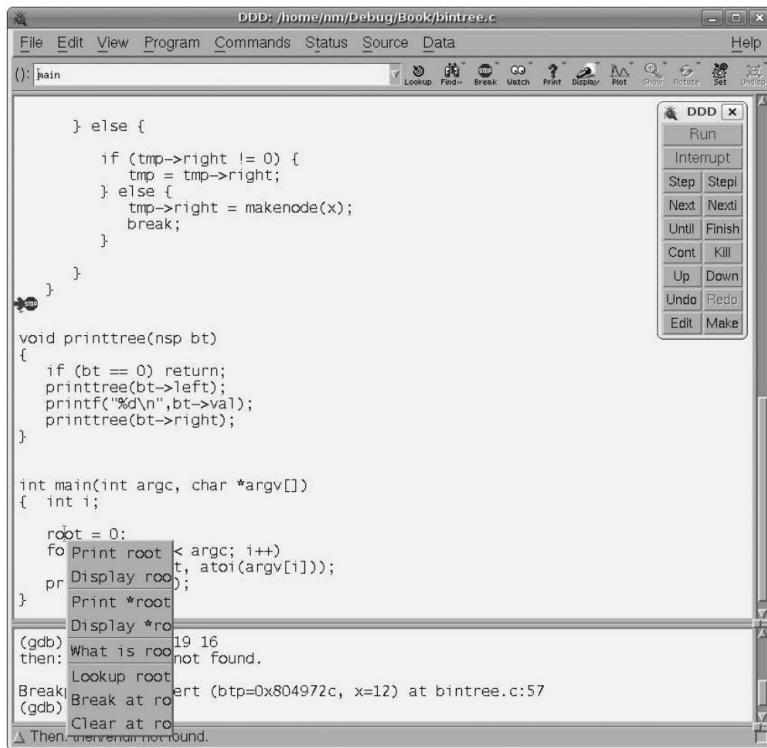


Рисунок3-1: Просмотр всплывающего окна для переменной

\$ ddd --separate bintree

затем появятся отдельные окна — Исходный код, Консоль и Данные, размер которого вы можете изменить по своему желанию.

Если вы хотите удалить часть или все содержимое окна данных Во время сеанса DDD есть много способов сделать это. Например, вы можете щелкните правой кнопкой мыши по элементу и выберите опцию «Скрыть».

3.2.3 Инспекция в затмении

Как и в случае с DDD, чтобы проверить скалярную переменную в Eclipse, просто переместите мышь указатель на любой экземпляр переменной в окне исходного кода. Обратите внимание, что Например, это должен быть независимый скаляр, а не скаляр внутри структуры . Это показано на рисунке 3-4. Мы успешно запросили значение x здесь, но если бы мы переместили указатель мыши на часть val в tmp->val в той же строке, это не скажет нам, что там.

На этом этапе вам следует воспользоваться представлением переменных Eclipse, которое вы можно увидеть в верхней правой части рисунка 3-5. Щелкните треугольник рядом с

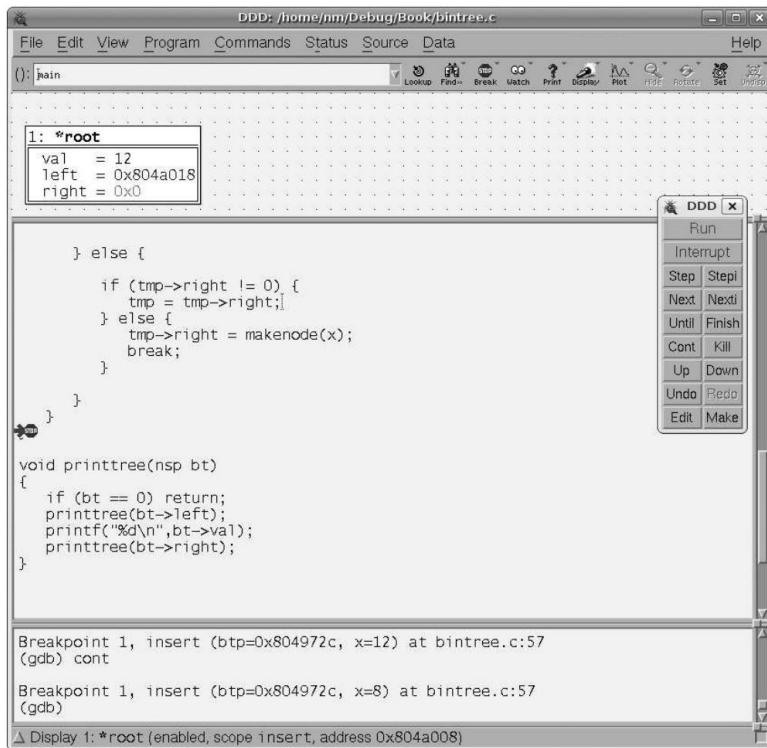


Рисунок 3-2: узел отображения

`tmp`, чтобы направить его вниз, затем прокрутите строку вниз и обнаружите, что `tmp->val` — это отображается. (Оказывается, их 12.)

И вы можете продолжить этот процесс. После нажатия на треугольник рядом с слева вы увидите экран, показанный на рисунке 3-6, где вы увидите, что `tmp->left->val` равен 8.

По умолчанию в представлении Переменные не отображаются глобальные переменные. В программе здесь есть одна, `root`. Вы можете добавить ее в представление Переменные, щелкнув правой кнопкой мыши в этом представлении, выбрав Добавить глобальные переменные, установив флажок для `root` в появившемся всплывающем окне, а затем нажмите кнопку OK.

3.2.4 Инспекция Динамичный Массивы

Как обсуждалось в Главе 1, в GDB вы можете распечатать весь массив, скажем, объявленный как

```
int x[25];
```

набрав

```
(гдб) пикс.
```

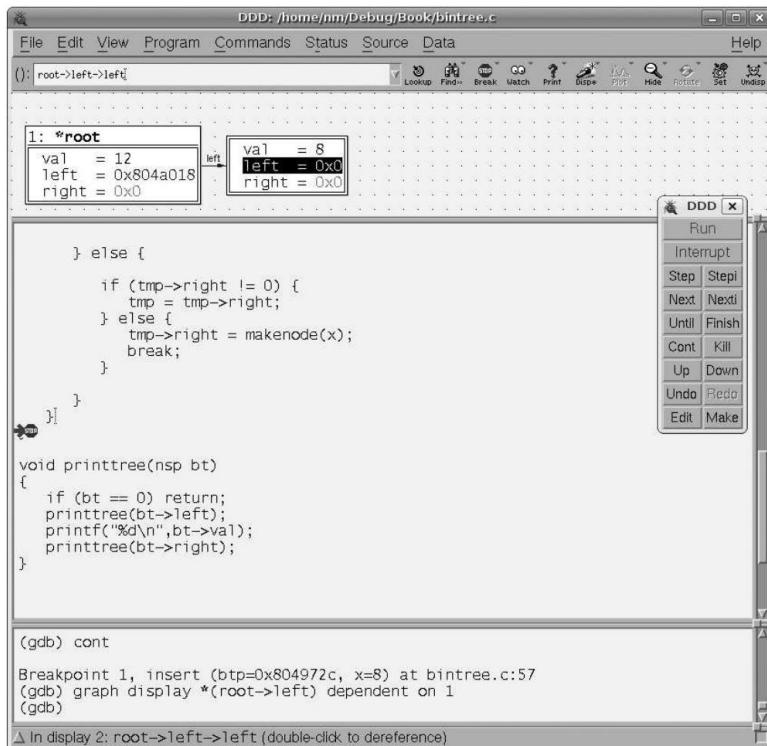


Рисунок 3-3: Перейдите по ссылкам

Но что, если бы массив был создан динамически, скажем, как

```
целое *x;
...
x = (целое число *) malloc(25*sizeof(целое число));
```

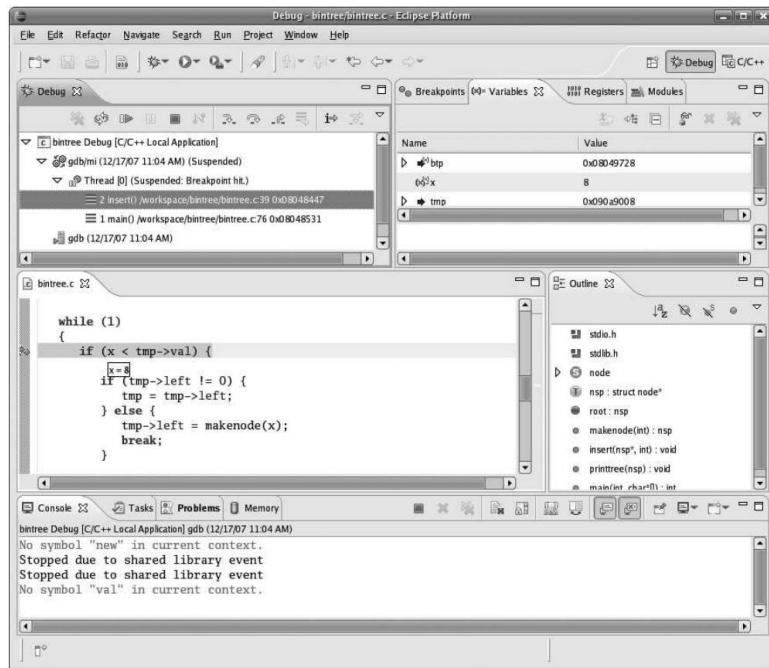
Если вы хотите распечатать массив в GDB, вы не можете ввести

(гdb) пикс.

Это просто выведет адрес массива. Вы также не сможете ввести

(гdb) п *x

Это выведет только один элемент массива, `x[0]`. Вы все еще можете распечатать отдельные элементы, как в команде `px[5]`, но вы не могли распечатайте весь массив, просто используя команду `print` на `x`.

Рисунок3-4: Инспекция α скалярнаяпеременная в Затмение

3.2.4.1 Решения в GDB

В GDB вы можете решить эту проблему, создав искусственный массив. Рассмотрим КОД

```

1 целое число *x;
2
3 основной()
4 {
5     x = (целое число *) malloc(25*sizeof(целое число));
6     x[3] = 12;
7 }
```

Тогда вы могли бы сделать что-то вроде этого:

```

Точка останова 1, main() в artif.c:6
6         x = (целое число *) malloc(25*sizeof(целое число));
(gdb) н
7         x[3] = 12;
(gdb) н
8
(gdb) п *x@25
$1 = {0, 0, 0, 12, 0 <повторяется 21 раз>}
```

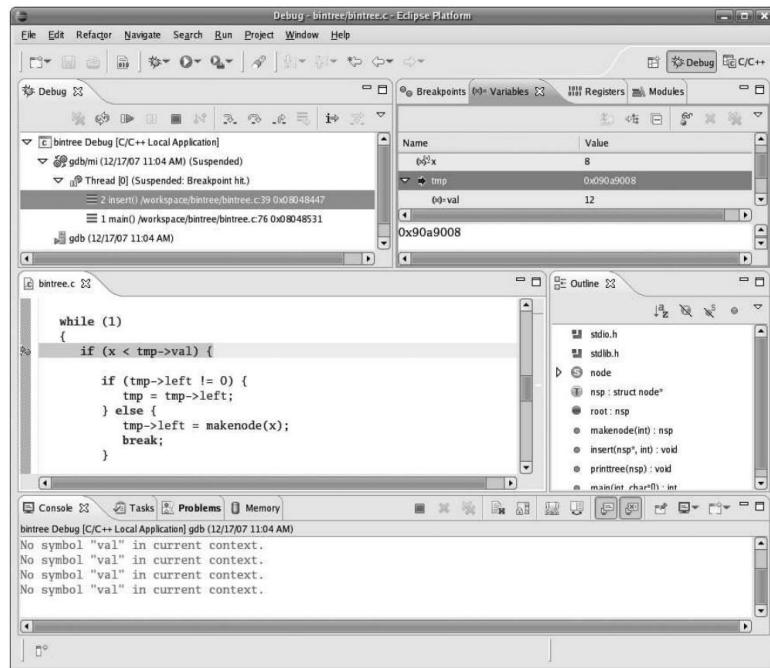


Рисунок3-5: Инспекция а структура поля в затмение

Как вы можете видеть, общая форма такова:

*указатель@число_элементов

GDB также позволяет использовать приведения типов, когда это уместно, например,

```
(gdb) p (int [25]) *x
$2 = {0, 0, 0, 12, 0 <повторяется 21 раз>}
```

3.2.4.2 Решения в DDD

Как всегда, вы можете использовать метод GDB, в данном случае искусственные массивы, через консоль DDD.

Другим вариантом может быть печать или отображение диапазона памяти (см. Раздел 3.2.7 ниже).

3.2.4.3 Решения в Eclipse

Здесь вы можете использовать команду Eclipse «Отобразить как массив».

Например, давайте немного расширим наш предыдущий пример:

1 целое число *x;

2

3 основной()

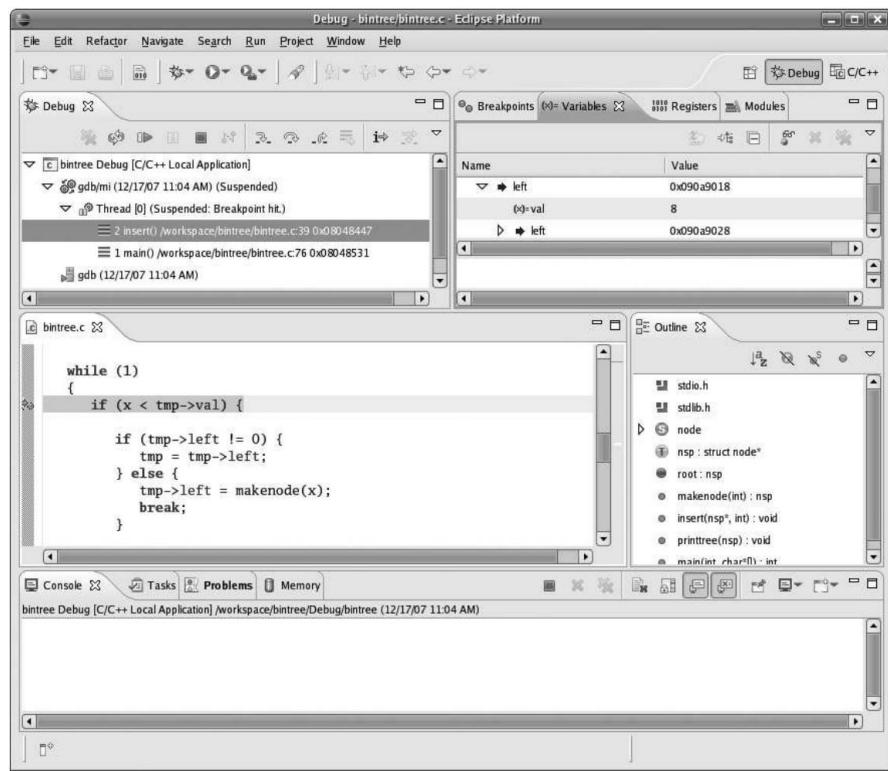


Рисунок 3-6: Следуя указателю ссылок в Затмение

```

4 {
5     целочисленный y;
6     x = (целое число *) malloc(25*sizeof(целое число));
7     scanf("%d%d",&x[3],&x[8]);
8     y = x[3] + x[8];
9     printf("%d\n",y);
10 }

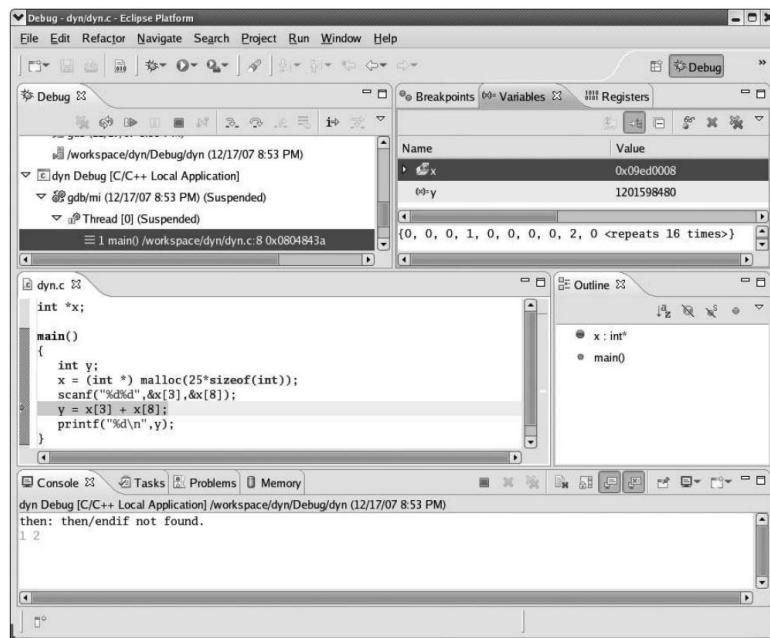
```

Допустим, вы сейчас на задании у. Сначала вы бы получили x в вид Переменные, щелкнув правой кнопкой мыши в этом виде и выбрав x. Вы бы затем снова щелкнули правой кнопкой мыши x в представлении «Переменные» и выберите «Отобразить как массив». В появившемся всплывающем окне необходимо заполнить поля Начальный индекс и Длина, скажем, с 0 и 25, чтобы отобразить весь массив. Экран теперь будет выглядеть так:

Рисунок 3-7. Вы можете увидеть массив в представлении «Переменные», показанный как

(0,0,0,1,0,0,0,0,2,0,<повторяется 16 раз>)

Значения 1 и 2 взяты из наших входных данных для программы, которые можно увидеть в Вид консоли.

Рисунок 3-7: Отображение ^a динамический в массиве Затмение

3.2.5 А как насчет C++?

Чтобы проиллюстрировать ситуацию для кода C++, вот версия двоичного кода C++
Пример дерева, использованный ранее:

```
// bintree.cc: процедуры для вставки и сортированной печати двоичного дерева в C++
```

```
#include <iostream.h>
```

```
класс узел {
```

```
общественно:
```

```
статический класс node *root; // корень всего дерева
```

```
int val; // сохраненное значение
```

```
узел класса *left; // указатель на меньший дочерний элемент
```

```
узел класса *right; // указатель на больший дочерний элемент
```

```
node(int x); // конструктор, устанавливающий val = x
```

```
static void insert(int x); // вставить x в дерево
```

```
static void printtree(class node *nptr); // распечатать поддерево с корнем *nptr
```

```
}
```

```
класс узел *node::root = 0;
```

```
узел::узел(целое x)
```

```

{
    значение
    = x; лево = право = 0;
}

узел void::вставка(int x) {

    если (узел::корень == 0)
        { узел::корень = новый узел(х);
        возврат;

    } класс узел *tmp=root;
    пока (1) {

        если (x < tmp->val) {

            если (tmp->left != 0) { tmp
                = tmp->left; } иначе
            { tmp->left
                = новый узел(х); break;

            }

        } еще {

            если (tmp->right != 0) { tmp
                = tmp->right; } иначе
            { tmp-
                >right = новый узел(х); break;

            }

        }

    }

}

void node::printtree(класс узел *np) {

    если (np == 0) return;
    node::printtree(np->left); cout <<
    np->val << endl;
    node::printtree(np->right);
}

```

```

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

    для (int i = 1; i < argc; i++)
        узел::вставка(atoi(argv[i]));
    узел::printtree(узел::корень);
}

```

Вы по-прежнему компилируете как обычно, убедившись, что компилятору нужно сохранить Таблица символов в исполняемом файле.²

Работают те же команды GDB, но с несколько иным выводом.

Например, снова распечатав содержимое объекта, на который указывает tmp внутри insert(), вы получите следующий вывод:

```
(gdb) p *tmp
$6 = {статический корень = 0x8049d08, значение = 19, левое = 0x0, правое = 0x0}
```

Это похоже на случай программы на языке C, за исключением того, что теперь также выводится значение статической переменной node::root (как и должно быть, поскольку она является частью класса).

Конечно, вам следует помнить, что GDB требует, чтобы вы указали переменные в соответствии с теми же правилами области видимости, которые использует C++. Например:

```
(gdb) p *корень
Невозможно получить доступ к памяти по
адресу 0x0 (gdb) p
*node::root $8 = {static root = 0x8049d08, val = 12, left = 0x8049d18, right = 0x8049d28}
```

Нам нужно было указать root через его полное имя, node::root. GDB и DDD не имеют встроенных обозревателей классов, но команда GDBptype удобна для быстрого просмотра структуры класса или структуры, например:

```
(gdb) ptype тип
узла = класс узла
{ public:
    static узел *root; int
    val; узел
    *left; узел
    *right;

    узел (целое);
    статическая пустота вставка (целое);
```

² Однако в связи с этим могут возникнуть различные проблемы. См. руководство GDB относительно форматов исполняемых файлов. В приведенных здесь примерах используется компилятор G++ (оболочка C++ для GCC) с опцией -g для указания того, что таблица символов должна быть сохранена. Мы также попробовали опцию -gstabs, которая сработала, но с несколько менее желательными результатами.

```
static void printtree(узел*);  
}
```

В DDD можно щелкнуть правой кнопкой мыши по имени класса или переменной, а затем выбрать что находится во всплывающем меню, чтобы получить ту же информацию.

В Eclipse есть режим Outline, поэтому вы можете легко получить информацию о классе таким образом.

3.2.6 Мониторинг Местные Переменные

В GDB вы можете получить список значений всех локальных переменных в текущем кадре стека, вызвав команду info locals .

В DDD вы даже можете отобразить локальные переменные, нажав Данные | Отобразить локальные переменные. Это приведет к разделу Окна данных DDD посвящено отображению местных жителей (обновляется по мере прохождения программы). Похоже, что в GDB нет прямого способа сделать это, хотя Вы можете сделать это на основе каждой точки останова, включив команду info locals в процедуру команд для каждой точки останова, в которой вы хотите автоматически распечатывать местные данные.

Как вы уже видели, Eclipse отображает локальные переменные в представлении «Переменные».

3.2.7 Рассматривание Память Напрямую

В некоторых случаях вам может понадобиться проверить память по указанному адресу, а не через имя переменной. GDB предоставляет команду x («examine») для этой цели. В DDD выбираем Данные | Память, указываем начальную точку и количество байтов, а также выбор между печатью и отображением. Eclipse имеет представление «Память», в котором можно создавать мониторы памяти.

Обычно это полезно в основном в контексте языка ассемблера и подробно обсуждается в Главе 8.

3.2.8 Расширенный для И Параметры Печать Отображать

Команды печати и отображения позволяют указывать альтернативные форматы. Например,

(гdb) п/ху

отобразит переменную у в шестнадцатеричном формате вместо десятичного. Другие часто используемые форматы: с для символа, s для строки и f для числа с плавающей точкой.

Вы можете временно отключить элемент отображения. Например,

(gdb) dis disp 1

временно отключает элемент 1 в списке отображения. Если вы не знаете элемент цифры, вы можете проверить через команду info disp . Чтобы снова включить элемент, используйте enable , например,

```
(gdb) включить disp 1
```

Чтобы полностью удалить отображаемый элемент, используйте команду `undisplay`, например:

```
(gdb) нераскрыть 1
```

3.3 Установка переменных из GDB/DDD/Eclipse

В некоторых случаях полезно задать значение переменной с помощью отладчика, в процессе выполнения программы. Таким образом, вы можете быстро ответить на вопросы «что если», которые возникают, когда вы выдвигаете гипотезы о различных источниках ошибки.

В GDB вы можете очень легко устанавливать значения, например,

```
(gdb) установить x = 12
```

изменит текущее значение x на 12.

Похоже, в DDD нет способа сделать это с помощью мыши, но опять же Вы можете выполнить любую команду GDB из DDD в окне консоли DDD.

В Eclipse перейдите в представление `Variables`, щелкните правой кнопкой мыши переменную, значение которой вы хотите установить, и выберите `Change Value`. Всплывающее окно затем позволит вам ввести новое значение.

Вы можете задать аргументы командной строки для вашей программы с помощью команды `GDB set args`. Однако это не дает никаких преимуществ по сравнению с методом, описанным в Главе 1, который просто использует новые аргументы при вызове команды `GDB run`. Эти два метода полностью эквивалентны. Например, это не тот случай, когда

```
(gdb) установить аргументы 1 52 19 11
```

немедленно изменит `argv[1]` на 1, `argv[2]` на 52 и т. д. Эти изменения не произойдут до следующего раза, когда вы дадите команду `run`.

В GDB есть команда `info args`, которую можно использовать для проверки аргументов текущей функции. DDD предоставляет ее, когда вы нажимаете `Данные | Отобразить аргументы`.

3.4 Собственные переменные GDB

Помимо переменных, которые вы объявляете в своей программе, GDB также предоставляет механизмы для других.

3.4.1 Использование Изготовление Ценить История

Выходные значения команды печати GDB помечаются как \$1, \$2 и т. д., при этом эти величины в совокупности называются историей стоимости. Их можно использовать для создания ярлыков в будущих командах печати , которые вы будете выполнять.

Например, рассмотрим пример bintree из раздела 3.1. Часть Сеанс GDB может выглядеть так:

```
(gdb) p tmp->left
$1 = (структурный узел *) 0x80496a8
(gdb) p *(tmp->left)
$2 = {значение = 5, левое = 0x0, правое = 0x0}
(гдб) п *$1
$3 = {значение = 5, левое = 0x0, правое = 0x0}
```

Здесь произошло следующее: после того, как мы распечатали значение указателя, tmp->left и обнаружили, что он не равен нулю, мы решили распечатать, что этот указатель указали. Мы сделали это дважды, сначала обычным способом, а затем через история ценностей.

В этом третьем выпуске мы ссылались на \$1 в истории стоимости. Если бы у нас было не прибегая к обычной печати, мы могли бы воспользоваться специальной переменной истории значений \$:

```
(gdb) p tmp->left
$1 = (структурный узел *) 0x80496a8
(гдб) п *$
$2 = {значение = 5, левое = 0x0, правое = 0x0}
```

3.4.2 Удобство Переменные

Скажем, у вас есть переменная указателя р , которая в разное время указывает на разные узлы в связанном списке. Во время сеанса отладки вы можете захотеть запишите адрес конкретного узла, например, потому что вы хотите перепроверить значение в узле в различные моменты в процессе отладки. Первое когда р достигает этого узла, вы можете сделать что-то вроде

```
(gdb) установить $q = p
```

и с тех пор делать такие вещи, как

```
(гдб) п *$q
```

Переменная \$q здесь называется удобной переменной. Удобные переменные могут изменять значения в соответствии с правилами языка C. Например, достаточно, рассмотрите код

```
int w[4] = {12,5,8,29};
```

```
основной()
```

```
{
    ш[2] = 88;
}
```

В GDB вы можете сделать что-то вроде

```
Точка останова 1, main() в cv.c:7 w[2] =
```

```
7          88;
(gdb) н 8
```

```
} (gdb) установить $i
= 0 (gdb) pw[$i++]
$1 = 12
(gdb)
$2 = 5
(gdb)
$3 = 88
(gdb)
$4 = 29
```

Чтобы понять, что здесь произошло, вспомните, что если мы просто нажмем клавишу ENTER в GDB, не выдавая команду, GDB воспримет это как запрос на повторение последней команды. В сеансе GDB выше вы продолжали нажимать клавишу ENTER , что означало, что вы просили GDB повторить команду

```
(gdb) pw[$i++]
```

Это означало не только то, что значение будет выведено на печать, но и то, что вспомогательная переменная \$i будет увеличиваться.

ПРИМЕЧАНИЕ. Вы можете выбрать практически любое имя для удобной переменной, с некоторыми естественными примерами. Например, вы не можете иметь вспомогательную переменную с именем \$3, так как она зарезервирована для элементов в истории значений. Кроме того, вы не должны использовать имена регистров, если вы работаете на языке ассемблера. Например, для машин на базе Intel x86 одним из имен регистров является EAX, а в GDB он упоминается как \$eax; вы бы не хотели выбирать это имя для вспомогательной переменной, если бы вы работали на уровне языка ассемблера.

4

КОГДА ПРОГРАММА ВЫХОДИТ ИЗ СТРОЯ



Говорят, что C — это язык низкого уровня. Отчасти это означает, что большая часть управления памятью для прикладной программы

Реализация grammar остается на усмотрение программиста.

Хотя этот подход может быть весьма эффективным, он также накладывает большую ответственность на программиста.

Говорят также, что C — относительно небольшой язык и его легко выучить. Однако язык C мал, если не принимать во внимание типичную реализацию стандартной библиотеки C, которая огромна, и многие программисты считают язык C простым в использовании, но только до тех пор, пока они не столкнутся с указателями.

В общем случае ошибка программы может привести к одному из двух последствий:

- Это может привести к тому, что программа сделает то, что программист не намеревался делать. Такие ошибки часто возникают из-за недостатков в логике, как в программе сортировки чисел в Главе 3, где мы поместили узел в неправильную ветвь дерева. До сих пор мы концентрировались на этом типе ошибок.
- Это может привести к «бомбардировке» или «падению» программы. Эти ошибки часто связаны с неправильным обращением или неправильным использованием указателей. Это тип ошибок, с которыми мы будем иметь дело в этой главе.

4.1 Справочный материал: Управление памятью

Что на самом деле происходит, когда программа дает сбой? Мы объясним это здесь и показать, как это связано с поиском ошибки, вызывающей сбой.

4.1.1

Почему программа аварийно завершает работу?

На языке мира программирования программа падает, когда ошибка приводит к тому, что она прекращает выполняться внезапно и ненормально. На сегодняшний день наиболее распространенной причиной сбоя является попытка программы получить доступ к памяти Местоположение без разрешения на это. Аппаратное обеспечение будет чувствовать это и выполнить переход к операционной системе (ОС). На платформах семейства Unix, которые находятся в центре нашего внимания здесь и в большей части этой книги, ОС обычно объявляет, что программа вызвала ошибку сегментации, обычно называемую ошибкой сегментации, и прекращает выполнение программы.

В системах Microsoft Windows соответствующий термин — общая защита.

Неисправность. Как бы ни называлось, оборудование должно поддерживать виртуальную память и ОС должна использовать его, чтобы произошла эта ошибка. Хотя это стандарт для современных компьютеров общего назначения, читатель должен иметь в виду имейте в виду, что это часто не относится к небольшим специализированным компьютерам, таким как как встроенные компьютеры, используемые для управления машинами.

Для эффективного использования GDB/DDD для устранения ошибок сегментации важно точно понимать, как возникают ошибки доступа к памяти. В следующих нескольких страницы, мы представим краткий урок о роли виртуальной памяти (VM) во время выполнения программ. Наше особое внимание будет удалено тому, как Проблемы VM связаны с ошибками сегментации. Таким образом, даже если вы изучали VM на курсах по вычислительной технике, фокус здесь может дать вам некоторые новые идеи, которые помогут вам устранийте ошибки сегментов в ходе отладки.

4.1.2

Программа Макет в Память

Как упоминалось ранее, ошибка сегмента возникает, когда ваша программа имеет память Проблема доступа. Чтобы обсудить это, важно сначала понять, как программа размещается в памяти.

На платформах Unix набор выделенных виртуальных адресов программы обычно имеет вид, подобный схеме на рисунке 4-1.

Здесь виртуальный адрес 0 находится внизу, а стрелки показывают направление роста двух компонентов, кучи и стека, пожирающих свободную область по мере их роста. Роли различных частей следующие:

- Текстовый раздел состоит из машинных инструкций, созданных компилятором из исходного кода вашей программы. Каждая строка кода C, для примера, обычно преобразуется в две или три машинные инструкции, и совокупность всех полученных инструкций составляет текстовую секцию исполняемого файла. Формальное название этой секции — .text.

Этот компонент включает статически связанный код, включая /usr/lib/crt0.o, системный код, который выполняет некоторую инициализацию, а затем вызывает ваш основной().



Рисунок4-1: Память программ макет

- Раздел данных содержит все переменные программы, которые выделяются во время компиляции, то есть ваши глобальные переменные.

На самом деле, этот раздел состоит из различных подразделов. Первый называется .data и состоит из ваших инициализированных переменных, то есть тех, которые указаны в объявлении типа

целое число x = 5;

Также существует раздел .bss для неинициализированных данных, указанных в объявлении типа

целочисленный y;

- Когда ваша программа запрашивает дополнительную память у операционной системы система во время выполнения — например, когда вы вызываете malloc() в C или вызываете новую конструкцию в C++ — запрошенная память выделяется в области, называемой кучей. Если у вас заканчивается место в куче, вызов brk() может быть использован для расширения кучи (именно это и делают malloc() и его друзья).
- Раздел стека — это пространство для динамически выделяемых данных. Данные для вызовов функций, включая аргументы, локальные переменные и адреса возврата, хранятся в стеке. Стек увеличивается каждый раз, когда выполняется вызов функции, и уменьшается каждый раз, когда функция возвращается к вызывающей стороне.
- Динамически связанный код вашей программы не показан на рисунке выше из-за платформозависимости его расположения, но он где-то там есть.

Давайте немного разберемся. Рассмотрим следующий код:

```
int q[200];
```

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

```

int i, n, *p;
p = malloc(sizeof(int));
scanf("%d", &n);
для (i = 0; i < 200; i++)
    q[i] = i;

printf("%x %x %x %x %x\n", main, q, p, &i, scanf);

возврат 0;
}

```

Программа сама по себе не делает многое, но мы написали ее как инструмент для неформального исследования макета виртуального адресного пространства. Для этого давайте запустим это:

```
% a вне
5
80483f4 80496a0 9835008 bfb3abec 8048304
```

Вы можете видеть, что приблизительное расположение текстового раздела, раздела данных-функции `tion`, `heap`, `stack` и динамически связанные функции — `0x080483f4`, `0x080496a0`, `0x09835008`, `0xbfb3abec` и `0x08048304` соответственно.

Вы можете получить точный отчет о структуре памяти программы в Linux посмотрев на файл карт процесса . Номер процесса оказывается 21111, поэтому посмотрим на соответствующий файл `/proc/21111/maps`:

```
$ кот /proc/21111/карты
009f1000-009f2000 r-xp 009f1000 00:00 0 [vds]
009f2000-00a0b000 r-xp 00000000 08:01 4116750 /lib/ld-2.4.so
00a0b000-00a0c000 r-xp 00018000 08:01 4116750 /lib/ld-2.4.so
00a0c000-00a0d000 rwxp 00019000 08:01 4116750 /lib/ld-2.4.so
00a0f000-00b3c000 r-xp 00000000 08:01 4116819 /lib/libc-2.4.so
00b3c000-00b3e000 r-xp 0012d000 08:01 4116819 /lib/libc-2.4.so
00b3e000-00b3f000 rwxp 0012f000 08:01 4116819 /lib/libc-2.4.so
00b3f000-00b42000 rwxp 00b3f000 00:00 0
08048000-08049000 r-xp 00000000 00:16 18815309 /home/matloff/a.out
08049000-0804a000 rw-p 00000000 00:16 18815309 /home/matloff/a.out
09835000-09856000 ч-п 09835000 00:00 0 [куча]
b7ef8000-b7ef9000 ч-п b7ef8000 00:00 0
b7f14000-b7f16000 rw-p b7f14000 00:00 0
fbfb27000-fbfb3c000 rw-p fbfb27000 00:00 0 [куча]
```

Вам не нужно понимать все это. Дело в том, что в этом представлении вы можете видеть ваши текстовые и секции данных (из файла `a.out`), а также кучу и стек. Вы также можете увидеть, где библиотека C (для вызовов `scanf()`, `malloc()`, и `printf()`) был помещен (из файла `/lib/libc-2.4.so`). Вы также должны распознать поле разрешений, формат которого похож на знакомый формат файла

миссии, отображаемые ls , указывающие привилегии , например, rw-r . Последнее будет объяснено в ближайшее время.

4.1.3 The Понятие страниц

Виртуальное адресное пространство, показанное на рисунке 4-1, концептуально простирается от 0 до $2^w - 1$, где w — размер слова вашей машины в битах. Конечно, ваша программа обычно будет использовать только малую часть этого пространства, и ОС может зарезервировать часть пространства для своей собственной работы. Но ваш код с помощью указателей может генерировать адрес в любом месте этого диапазона. Часто такие адреса будут неверными из-за «энтомологических условий», то есть из-за ошибок в вашей программе!

Это виртуальное адресное пространство рассматривается как организованное в куски, называемые страницами. На оборудовании Pentium размер страницы по умолчанию составляет 4096 байт. Физическая память (как ОЗУ, так и ПЗУ) также рассматривается как разделенная на страницы. Когда программа загружается в память для выполнения, ОС организует хранение некоторых страниц программы на страницах физической памяти. Эти страницы называются резидентными, а остальные хранятся на диске.

В разное время во время выполнения некоторые страницы программы, которые не являются текущими, будет необходим резидентный резидент. Когда это произойдет, это будет воспринято оборудованием, которое передаст управление ОС. Последняя перенесет требуемую страницу в память, возможно, заменив другую страницу программы, которая в данный момент является резидентной (если нет свободных страниц памяти), а затем вернет управление нашей программе. Вытесненная страница программы, если таковая имеется, становится нерезидентной и будет сохранена на диске.

Чтобы управлять всем этим, ОС поддерживает таблицу страниц для каждого процесса. (Таблицы страниц Pentium имеют иерархическую структуру, но здесь для простоты мы предполагаем только один уровень, и большая часть этого обсуждения не будет касаться Pentium.) Каждая из виртуальных страниц процесса имеет запись в таблице, которая включает следующую информацию:

- Текущее физическое местоположение этой страницы в памяти или на диске. В последнем случае запись будет указывать на то, что страница нерезидентная и может состоять из указателя на список, который в конечном итоге ведет к физическому местоположению на диске. Она может показывать, например, что виртуальная страница 12 программы является резидентной и расположена на физической странице 200 памяти.
- Разрешения — чтение, запись, выполнение — для этой страницы.

Обратите внимание, что ОС не будет выделять частичные страницы для программы. Например, если программа, которую нужно запустить, имеет общий размер около 10 000 байт, она займет три страницы памяти при полной загрузке. Она не просто займет около 2,5 страниц, поскольку страницы являются наименьшей единицей памяти, которой манипулирует система виртуальной машины. Это важный момент для понимания при отладке, поскольку он подразумевает, что некоторые ошибочные доступы к памяти программой не вызовут сбоев сегментации, как вы увидите ниже. Другими словами, во время сеанса отладки вы не можете сказать что-то вроде: «Эта строка исходного кода, должно быть, в порядке, поскольку она не вызвала сбоя сегментации».

4.1.4 Подробности Роль страницы Стол

Имейте в виду виртуальное адресное пространство в Таблице 4-1 и продолжайте считать, что размер страницы составляет 4096 байт. Тогда виртуальная страница 0 будет содержать байты 0 по 4095 виртуального адресного пространства, страница 1 содержит байты 4096 до 8191 и так далее.

Как уже упоминалось, когда мы запускаем программу, ОС создает таблицу страниц, которую она использует для управления виртуальной памятью процесса, который выполняет программу код. (Обзор процессов ОС представлен в материале о потоках в Глава 5.) Всякий раз, когда этот процесс запускается, регистр таблицы страниц оборудования будет указывать на эту таблицу.

Концептуально говоря, каждая страница виртуального адресного пространства процесса имеет запись в таблице страниц (на практике для этого можно использовать различные приемы). скажи таблицу). Эта запись таблицы страниц хранит различные фрагменты информации, относящиеся к странице. Данные, представляющие интерес в отношении ошибок сегмента, являются правами доступа к странице, которые аналогичны правам доступа к файлам: читать, писать и выполнять. Например, запись таблицы страниц для страницы 3 будет указать, имеет ли ваш процесс право читать данные с этой страницы, право записывать на него данные и право выполнять на нем инструкции (если страница содержит машинный код).

В процессе выполнения программы будет постоянно обращаться к различным своим разделам, описанный выше, который заставляет оборудование обращаться к таблице страниц следующим образом:

- Каждый раз, когда программа использует одну из своих глобальных переменных, доступ на чтение/запись в раздел данных обязательно.
- Каждый раз, когда программа обращается к локальной переменной, программа обращается стек, требующий доступа для чтения/записи к разделу стека.
- Каждый раз, когда программа входит в функцию или выходит из нее, она делает один или несколько доступ к стеку, требующий доступа для чтения/записи к разделу стека.
- Каждый раз, когда программа обращается к хранилищу, которое было создано вызовом `malloc()` или `new` происходит доступ к куче, что снова требует доступа на чтение/запись.
- Каждая машинная инструкция, которую выполняет программа, будет извлечена из текстового раздела (или из области для динамически связанного кода), таким образом, требуется разрешение на чтение и выполнение.

Во время выполнения программы генерируемые ею адреса будут виртуальный. Когда программа пытается получить доступ к памяти по определенному виртуальному адресу, скажем, у, оборудование преобразует его в виртуальный номер страницы v , который равен u , деленное на 4096 (где деление использует целочисленную арифметику, отбрасывая остаток). Затем оборудование проверит запись v на странице таблицу, чтобы увидеть, соответствуют ли разрешения для страницы операции быть выполнены. Если они совпадают, оборудование получит желаемое местоположение фактический номер физической страницы из этой записи таблицы, а затем выполнить запрошенную операцию памяти. Но если запись таблицы показывает, что запрошеннная операция не имеет соответствующего разрешения, оборудование выполнит внутреннее прерывание. Это вызовет переход к процедуре обработки ошибок ОС. Обычно ОС затем объявляет о нарушении доступа к памяти и

прекратить выполнение программы (т. е. удалить ее из таблицы процессов и из памяти).

Ошибка в вашей программе может привести к несоответствию прав доступа и сгенерировать ошибку сегмента во время любого из типов доступа к памяти, перечисленных выше. Например, предположим, что ваша программа содержит глобальное объявление

```
int x[100];
```

и предположим, что ваш код содержит оператор

```
x[y] = 3;
```

Вспомним, что в C/C++ выражение $x[i]$ эквивалентно (и на самом деле означает) $*(x+i)$, то есть содержимое ячейки памяти, на которую указывает адрес $x+i$. Если смещение i равно, скажем, 200000, то это, скорее всего, создаст адрес виртуальной памяти у , который находится за пределами набора страниц, назначенных ОС для раздела данных программы, где компилятор и компоновщик организовали хранение массива $x[]$. Тогда при попытке операции записи возникнет ошибка сегмента.

Если бы x был локальной переменной, то та же проблема возникла бы в разделе стека.

Нарушения, связанные с разрешением на выполнение, могут происходить и более тонкими способами. В программе на языке ассемблера, например, у вас может быть элемент данных с именем `sink` и функция с именем `sunk()`. При вызове функции вы можете случайно написать

```
вызов раковина
```

вместо

```
вызов затонул
```

Это приведет к ошибке сегмента, поскольку программа попытается выполнить инструкцию по адресу приемника, который находится в разделе данных, а страницы раздела данных не имеют разрешений на выполнение.

Точный аналог этой ошибки кодирования не приведет к ошибке сегментации в языке C, поскольку компилятор будет возражать против такой строки:

```
z = раковина(5);
```

когда `sink` был объявлен как переменная. Но эта ошибка может легко возникнуть, когда используются указатели на функции. Рассмотрим такой код:

```
целочисленный fцелочисленный
x) {
    вернуть x*x;
}
```

целое (*p)(целое);

```
int main(пустота)
{
    p = x;
    и=(*п)(5);
    printf("%d\n", u);

    возврат 0;
}
```

Если бы вы забыли утверждение `p = f;` тогда `p` было бы равно 0, и вы попытается выполнить инструкции, находящиеся на странице 0, странице, для которой вы не будет иметь разрешения на выполнение (или иного) для (вспомните рисунок 4-1).

4.1.5 Небольшое Доступ к памяти Ошибка Мощь Нет Причина а Сегмент Вина

Чтобы углубить свое понимание того, как возникают сбои сегментов, рассмотрите следующий код, поведение которого при выполнении показывает, что ошибки сегмента не возникают всегда происходят в ситуациях, когда можно было бы ожидать, что они:

```
int q[200];

основной()
{
    целочисленный i;
    для (i = 0; i < 2000; i++) {
        q[i] = i;
    }
}
```

Обратите внимание, что программист, по-видимому, допустил опечатку. в цикле, устанавливая 2000 итераций вместо 200. Компилятор C будет не отловить это во время компиляции, и машинный код, сгенерированный Во время выполнения компилятор проверяет, не выходит ли индекс массива за пределы допустимого диапазона. (Это значение по умолчанию в GCC, хотя он также предлагает опцию `-fmudflap`, которая делает это (Предоставьте такую проверку индекса во время выполнения.)

Во время выполнения вполне вероятно возникновение ошибки сегмента. Однако время-
ин ошибки может вас удивить. Ошибка вряд ли появится на «естественное» время, то есть когда `i = 200`; скорее всего, это произойдет гораздо позже, чем это.

Чтобы проиллюстрировать это, мы запустили эту программу на ПК с Linux под GDB, в для того, чтобы удобно запрашивать адреса переменных. Оказалось, что сегмент Ошибка произошла не при `i = 200`, а при `i = 728`. (Ваша система может выдавать другие значения) (Результаты будут, но принципы будут теми же.) Давайте посмотрим, почему.

Из запросов к GDB мы обнаружили, что массив `q[]` заканчивается по адресу `0x80497bf`; то есть последний байт `q[199]` был в этом месте памяти. Взяв принимая во внимание размер страницы Intel в 4096 байт и размер слова в 32 бита

машине виртуальный адрес разбивается на 20-битный номер страницы и 12-битный битовое смещение. В нашем случае $q[0]$ закончилось на виртуальной странице с номером $0x8049 = 32841$, смещение $0x7bf = 1983$. Таким образом, на странице все еще было $4096 - 1984 = 2112$ байт памяти, на которой был выделен q . Это пространство может содержать $2112 / 4 = 528$ целочисленных переменных (поскольку каждая имеет ширину 4 байта на используемой здесь машине), и наш код обработал его так, как будто он содержал элементы q в «позициях» от 200 до 727.

Конечно, эти элементы $q[0]$ не существуют, но компилятор их не обнаружил. Жалобы не поступало и с оборудованием, поскольку записи все еще выполнялись на страницу, на которую у нас, безусловно, было разрешение на запись (потому что некоторые фактических элементов $q[0]$ лежат на нем, и $q[0]$ выделяется в сегменте данных). Только когда я стал 728, $q[0]$ ссылался на адрес на другом страница. В данном случае это была страница, для которой у нас не было записи (или любой другой) разрешение; оборудование виртуальной памяти обнаружило это и запустило сегмент вина.

Поскольку каждая целочисленная переменная хранится в 4 байтах, эта страница содержит 528 ($2112 / 4$) дополнительных «фантомных» элементов, которые код обрабатывает как принадлежащие массиву $q[0]$. Таким образом, хотя мы и не подразумевали, что это должно быть сделано, все еще допустимо получить доступ к $q[200]$, $q[201]$ и так далее, вплоть до элемента $199 + 528 = 727$, то есть, $q[727]$ — без срабатывания ошибки сегмента! Только когда вы пытаетесь получить доступ к $q[728]$, вы сталкиваетесь с новой страницей, для которой вы может иметь или не иметь требуемые разрешения на доступ. Здесь мы не имели, и так что программа сегмента дала сбой. Однако следующая страница может, по чистой случайности, на самом деле имели бы надлежащие привилегии, назначенные ему, и тогда не было бы было еще больше фантомных элементов массива.

Мораль: как уже говорилось ранее, мы не можем делать вывод из отсутствия ошибки сегмента, указывающая на правильность операции с памятью.

4.1.6 Сегмент Неисправности и Unix Сигналы

В обсуждении выше мы сказали, что ошибка сегмента обычно приводит к завершению программы. Это верно, но для серьезной отладки есть немного больше того, о чем вам следует знать в связи с сигналами Unix.

Сигналы указывают на исключительные условия и передаются во время выполнения программы, позволяя ОС (или вашему собственному коду) реагировать на различные ситуации событий. Сигнал может быть подан в процессе базовым аппаратным обеспечением системы (как с SIGSEGV или SIGFPE), операционной системой (как SIGTERM или SIGABRT), или другим процессом (как с SIGUSR1 или SIGUSR2), или он может быть даже отправлен самим процессом (через вызов библиотеки raise()).

Самый простой пример сигнала — нажатие CTRL-C на вашем клавиатуре во время работы программы. Нажатие (или отпускание) любой клавиши на ваша клавиатура генерирует аппаратное прерывание, которое вызывает процедуру ОС запустить. Когда вы нажимаете CTRL-C, ОС распознает эту комбинацию клавиш как специальный шаблон и выдает сигнал SIGINT для процесса на управляющем терминале. В просторечии говорят, что ОС «посыпает сигнал процесс». Мы будем использовать эту фразу, но важно понимать, что на самом деле ничего не «отправляется» в процесс. Все, что происходит, это то, что ОС записывает сигнал в своей таблице процессов, так что в следующий раз, когда процесс получит сигнал

сигнал получает квант времени на ЦП, будет выполнена соответствующая функция обработчика сигнала, как описано ниже. (Однако, учитывая предполагаемую срочность сигналов, ОС также может принять решение предоставить принимающему процессу следующий квант времени раньше, чем это было бы в противном случае.)

Существует множество различных типов сигналов, которые могут быть вызваны процессом. В Linux вы можете просмотреть весь список сигналов, введя команду

сигнал человека 7

в командной строке оболочки. Сигналы определены в различных стандартах, таких как POSIX.1, и эти сигналы будут присутствовать во всех операционных системах, которые соответствуют. Существуют также сигналы, которые являются уникальными для отдельных операционных систем.

Каждый сигнал имеет свой собственный обработчик сигнала, который является функцией, которая вызывается, когда этот конкретный сигнал поднимается в процессе. Возвращаясь к нашему примеру CTRL-C, когда поднимается SIGINT, ОС устанавливает текущую инструкцию процесса в начало обработчика сигнала для этого конкретного сигнала.

Таким образом, когда процесс возобновится, он выполнит обработчик.

Для каждого типа сигнала существует обработчик сигналов по умолчанию, что освобождает вас от необходимости писать их самостоятельно, если в этом нет необходимости.

Большинство безвредных сигналов по умолчанию игнорируются. Более серьезные типы сигналов, например, возникающие из-за нарушения прав доступа к памяти, указывают на условия, которые делают нецелесообразным или даже невозможным продолжение выполнения программы. В таких случаях обработчик сигналов по умолчанию просто завершает программу.

Некоторые обработчики сигналов не могут быть переопределены, но во многих случаях вы можете написать свой собственный обработчик, чтобы заменить обработчик по умолчанию, предоставляемый ОС. В Unix это делается с помощью системных вызовов signal() или sigaction().¹ Ваша пользовательская функция обработчика может, например, игнорировать сигнал или даже попросить пользователя выбрать курс действий.

Просто ради развлечения мы написали программу, которая иллюстрирует, как можно написать свой собственный обработчик сигнала и, используя signal(), вызвать или переопределить обработчик ОС по умолчанию или проигнорировать сигнал. Мы выбрали SIGINT, но вы можете сделать то же самое для любого сигнала, который может быть перехвачен. Программа также демонстрирует, как используется raise() .

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

```
void my_sigint_handler( int signum ) {
```

```
    printf("Я получил сигнал %d (для вас это 'SIGINT').\n", signum); puts("Хи-
хи! Щекотно!\n");
```

¹ Есть две функции, которые используются для переопределения обработчиков сигналов по умолчанию, поскольку Linux, как и другие Unix, соответствует нескольким стандартам. Функция signal(), которая проще в использовании, чем sigaction(), соответствует стандарту ANSI, тогда как функция sigaction() сложнее, но и более универсальна и соответствует стандарту POSIX.

```

}

целочисленный основной(void)
{
    char выборстр[20]; int
    выбор;

    в то время как ( 1 )
    {
        puts("1. Игнорировать control-C");
        puts("2. Пользовательский дескриптор
control-C"); puts("3. Использовать обработчик по
умолчанию control-C"); puts("4. Вызвать
SIGSEGV на себя."); printf("Введите свой выбор: ");

        fgets(choicestr, 20, stdin);
        sscanf(choicestr, "%d", &choice);

        если ( выбор == 1 )
            сигнал(SIGINT, SIG_IGN); // Игнорировать control-C
        иначе если (выбор == 2)
            сигнал(SIGINT, my_sigint_handler); иначе
        если (выбор == 3)
            сигнал(SIGINT, SIG_DFL);
        иначе если (выбор == 4)
            raise(SIGSEGV); иначе

        puts("Как скажешь, шеф.\n\n");
    }

    возврат 0;
}

```

Когда программа совершает нарушение доступа к памяти, в процессе возникает сигнал SIGSEGV. Обработчик ошибок сегмента по умолчанию завершает процесс и записывает «файл ядра» на диск, что мы вскоре объясним.

Если вы хотите сохранить программу в рабочем состоянии, вместо того, чтобы позволить ей завершиться, вы можете написать специальный обработчик для SIGSEGV. Действительно, вы можете захотеть намеренно вызвать ошибки сегментации, чтобы выполнить какую-то работу. Например, некоторые пакеты программного обеспечения для параллельной обработки используют искусственные ошибки сегментации, на которые реагирует специальный обработчик, чтобы поддерживать согласованность между различными узлами системы, как вы увидите в Разделе 5. Другое использование специализированных обработчиков для SIGSEGV, которое будет обсуждаться в Главе 7, включает инструменты для обнаружения и корректного реагирования на ошибки сегментации.

Однако пользовательские обработчики сигналов могут вызывать осложнения при использовании GDB/DDD/Eclipse. Независимо от того, используется ли он сам по себе или через DDD GUI,

GDB останавливает процесс при появлении любого сигнала. В случае приложений которые работают как упомянутое выше программное обеспечение параллельной обработки, это означает, что GDB будет останавливаться очень часто по причинам, не связанным с вашей отладкой работы. Чтобы справиться с этим, вам нужно будет сказать GDB не останавливаться, когда определенные сигналы возникают при использовании команды `handle`.

4.1.7 Другой Типы из Исключений

Существуют и другие источники сбоев, помимо ошибок сегментации. Исключения с плавающей точкой (FPE) вызывают сигнал `SIGFPE`. Хотя это называется исключение «плавающей точки», этот сигнал охватывает исключения целочисленной арифметики а также, как переполнение и условия деления на ноль. В системах GNU и BSD обработчикам FPE передается второй аргумент, который дает причину FPE. Обработчик по умолчанию будет игнорировать `SIGFPE` при некоторых обстоятельствах, например, переполнение числа с плавающей точкой, и завершить процесс в других обстоятельствах, например, при делении целочисленного числа на ноль.

Ошибка шины возникает, когда ЦП обнаруживает аномальное состояние нашине.

его шина при выполнении машинных инструкций. Различные архитектуры имеют различные требования к тому, что должно происходить в автобусе, и Точная причина аномалии зависит от архитектуры. Некоторые примеры Ситуации, которые могут вызвать ошибку шины, включают в себя следующее:

- Доступ к физическому адресу, который не существует. Это отличается от ошибки сегмента, поскольку ошибка сегмента подразумевает доступ к памяти, для которой нет Недостаточно привилегий. Сбои сегмента — это вопрос разрешений; ошибки шины являются вопросом предоставления процессору недействительного адреса.
- Во многих архитектурах машинные инструкции, которые обращаются к 32-битным числам, связи должны быть выровнены по словам, что означает, что адрес памяти количество должно быть кратно 4. Ошибка указателя, которая приводит к попытке доступа к 4-байтовому числу по нечетному адресу, может вызвать ошибку шины:

```
int main(пустота)
{
    символ *char_ptr;
    int *int_ptr;
    int int_array[2];

    // char_ptr указывает на первый элемент массива
    char_ptr = (char *) int_array;

    // Заставляет int_ptr указывать на один байт дальше начала существующего int.
    // Поскольку int не может быть только одним байтом, int_ptr больше не выравнивается.
    int_ptr = (int *) (char_ptr+1);

    *int_ptr = 1;                                // И это может вызвать ошибку шины.
}
```

```
    возврат 0;
```

```
}
```

Эта программа не вызовет ошибку шины в Linux, работающей на архитектура x86, так как на этих процессорах невыровненная память доступы являются допустимыми; они просто выполняются медленнее, чем выровненные доступы.

В любом случае ошибка шины — это исключение на уровне процессора, которое вызывает Сигнал SIGBUS должен быть поднят в системе Unix. По умолчанию SIGBUS вызовет процесс сброса ядра и завершения.

4.2 Основные файлы

Как упоминалось ранее, некоторые сигналы указывают на то, что продолжение процесса нецелесообразно или даже невозможно. В этих случаях действие по умолчанию заключается в преждевременном завершении процесса и записи файла, называемого файлом ядра, в разговорной речи известный как дамп ядра. Запись файлов ядра может быть подавлена вашим оболочкой (подробности см. в разделе 4.2.2).

Если основной файл создается во время выполнения вашей программы, вы можете открыть ваш отладчик, например GDB, для этого файла, а затем продолжите работу с вашим обычным GDB операции.

4.2.1 Как Основной Файлы являются Созданный

Основной файл содержит подробное описание состояния программы, когда она died: содержимое стека (или, если программа многопоточная, стеки для каждого потока), содержимое регистров ЦП (опять же с одним набором регистрировать значения на поток, если программа многопоточная), значения статически выделенные переменные программы (глобальные и статические переменные) и скоро.

Создать файл ядра очень просто. Вот код, который его генерирует:

```
int main(пустота)
{
    прервать();

    возврат 0;
}
```

Листинг 4-1: abort.c

Функция abort() заставляет текущий процесс получить сигнал SIGABRT , а обработчик сигнала по умолчанию для SIGABRT завершает программу и делает дамп ядра. Вот еще одна короткая программа, которая делает дамп ядра. В этой программе мы намеренно разыменовываем указатель NULL:

```
int main(пустота)
{
```

```

символ *c = 0;
printf("%s\n", *c);

возврат 0;
}

```

Листинг 4-2: sigsegv.c

Давайте сгенерируем основной файл. Скомпилируем и запустим sigsegv.c:

```
$ gcc -g -W -Wall sigsegv.c -o sigsegv
```

```
$ ./sigsegv
```

```
Ошибка сегментации (сброс керна)
```

Если вы просмотрите текущий каталог, вы увидите новый файл с именем core (или какой-то его вариант). Когда вы видите файл core где-то в вашей файловой системе, может быть не очевидно, какая программа его сгенерировала. Команда Unix файл умело сообщает нам имя исполняемого файла, который выгрузил этот конкретный основной файл:

```
$ файл core
```

```
ядро: ELF 32-битный LSB-файл ядра Intel 80386, версия 1 (SYSV), стиль SVR4,
```

```
Стиль SVR4, от 'sigsegv'
```

Основные соглашения об именовании файлов

Соглашение об именовании основных файлов раньше было простым: они все назывались ядром.

Затем, под GNU/Linux, многопоточные программы начали для дампа ядра с использованием имен файлов типа core.3928, где числовое значение часть имени файла указывает идентификатор процесса который сбросил ядро.

Начиная с ядра Linux 2.5, у вас есть контроль над именами, назначенными основным файлам с использованием интерфейса /proc/sys/kernel/. Механизм довольно прост и хорошо документирован в Documentation/sysctl/kernel.txt в дереве исходных кодов ядра Linux.

4.2.2 Твой Оболочка Может Подавить Создание из Основной Файл

Во многих, если не в большинстве случаев, процесс отладки не затрагивает основные файлы.

Если в программе происходит сбой сегмента, программист просто открывает отладчик, например GDB и снова запускает программу, чтобы воссоздать ошибку. По этой причине и

Поскольку файлы ядра, как правило, имеют большой размер, большинство современных оболочек имеют механизмы, которые изначально предотвращают запись файлов ядра.

В bash вы можете управлять созданием основных файлов с помощью команды ulimit :

```
ulimit -c n
```

где n — максимальный размер файла ядра в килобайтах. Любой файл ядра размером более nКБ не будет записан. Если вы не укажете n, оболочка отобразит текущее ограничение на файлы ядра. Если вы хотите разрешить файл ядра любого размера, вы можете использовать

```
ulimit -c неограниченный
```

Для пользователей tcsh и csh команда limit управляет размерами основных файлов. Например,

```
ограничение размера дампа ядра 1000000
```

сообщит оболочке, что вы не хотите создавать файл ядра, если его размер превысит миллион байт.

Если после запуска sigsegv вы не получили файл ядра, проверьте текущее ядро ограничения файлов с использованием ulimit -c для bash или limit -c для tcsh или csh.

Зачем вам вообще может понадобиться файл core? Поскольку вы можете просто перезапустить программу, в которой произошел сбой seg, из GDB и воссоздать сбой seg, зачем вообще беспокоиться о файлах core? Ответ в том, что в некоторых ситуациях, например, в следующих, это предположение не оправдано:

- Ошибка сегмента возникает только после того, как программа работает в течение длительного периода времени, поэтому воссоздать ошибку в отладчике невозможно.
- Поведение программы зависит от случайных событий окружающей среды, поэтому повторный запуск программы может не воспроизвести ошибку сегмента.
- Ошибка сегментации возникает, когда программа запускается неопытным пользователем. В этом случае пользователь, который обычно не является программистом (или не имеет доступа к исходному коду), не будет выполнять отладку. Однако такой пользователь все равно может отправить основной файл (если он был доступен) программисту для проверки и отладки.

Однако следует отметить, что если исходный код программы недоступен или исполняемый файл не был скомпилирован с использованием расширенной таблицы символов, или даже если мы не планируем отлаживать исполняемый файл, файлы ядра просто не очень полезны.

4.3 Расширенный пример

В этом разделе мы представляем подробный пример отладки неисправностей сегментов.

Ниже приведен код на языке C, который может быть частью реализации управляемого строкового типа, похожего на строки C++. Код, содержащийся в исходном файле cstring.c, реализует тип CString; однако он полон ошибок, как очевидных, так и скрытых. Наша цель — найти все эти ошибки и исправить их.

`CString` — это `typedefed` псевдоним для структуры, содержащей указатель на хранилище для строки символов вместе с переменной, которая хранит длину строки. Были реализованы некоторые вспомогательные функции, полезные для обработки строк:

`Init_CString()` Принимает в качестве аргумента строку С старого стиля и использует ее для инициализации новой `CString`.

`Delete_CString()` `CStrings` размещаются в куче, и их память должен быть освобожден, когда он больше не нужен. Эта функция занимается сборкой мусора.

`Chomp()` Удаляет и возвращает последний символ `CString`.

`Append_Chars_To_CString()` Добавляет строку в стиле С к `CString`.

Наконец, `main()` — это наша функция драйвера для тестирования реализации `CString`.

В нашем коде используется чрезвычайно полезная библиотечная функция `snprintf()`.

Если вы еще не сталкивались с этой функцией, она почти как `printf()`, за исключением того, что она записывает свои выходные данные в массив символов, а не в `stdout`.

Чтобы предотвратить переполнение буфера (которое может произойти с любой функцией, копирующей строки с завершающим нулем, если нулевой символ отсутствует в исходной строке), `snprintf()` также позволяет указать максимальное количество байтов для записи, включая завершающий нулевой символ:

```
#include <stdio.h>
#define STRSIZE 22

целочисленный основной(void)
{
    char s1[] = "тормоз"; char
    *s2 = "точки останова"; char
    logo[STRSIZE];

    sprintf(logo, STRSIZE, "%c %s %d %s.", 'I', s1, 2+2, s2);

    помещает(логотип);
    возвращает 0;
}
```

Эта программа запишет строку «Я останавливаю 4 точки останова» в массив символов логотипа, готовый к печати на наклейке на бампер.

А вот реализация нашего `CString`:

```
#include <stdio.h>
#include <stdlib.h>
#include <строка.h>

typedef struct { char
    *str; int len;
```

`} CString;`

```

CString *Init_CString(char *str) {

    CString *p = malloc(sizeof(CString)); p->len =
    strlen(str); strncpy(p->str,
    str, strlen(str) + 1); return p;

}

void Delete_CString(CString *p) {

    свободно(р);
    свободно(р->str);
}

// Удаляет последний символ CString и возвращает его. //

char Chomp(CString *cstring) {

    char lastchar = *( cstring->str + cstring->len );
    // Сокращаем строку на одну
    *( cstring->str + cstring->len ) = '0'; cstring->len =
    strlen( cstring->str );

    вернуть последний символ;
}

// Добавляет символ * к CString //

CString *Append_Chars_To_CString(CString *p, char *str) {

    char *newstr = malloc(p->len + 1); p->len =
    p->len + strlen(str);

    // Создаем новую строку для замены p->str
    sprintf(newstr, p->len, "%s%s", p->str, str);
    // Освобождаем старую строку и заставляем CString указывать на новую
    строку free(p-
    >str); p->str = newstr;

    вернуть p;
}

```

```

целочисленный основной(void)
{
    CString *mystr;
    СИМВОЛ С;

    mystr = Init_CString("Привет!");
    printf("Init:\n str: '%s' len: %d\n", mystr->str, mystr->len); c = Chomp(mystr);
    printf("Chomp '%c':
    \n str: '%s' len: %d\n", c, mystr->str, mystr->len); mystr = Append_Chars_To_CString(mystr,
    " world!"); printf("Append:\n str: '%s' len: %d\n", mystr->str,
    mystr->len);

    Delete_CString(mystr);

    возврат 0;
}

```

Изучите код и попробуйте угадать, что должно получиться на выходе. Затем скомпилируйте и запустите его.

```
$ gcc -g -W -Wall cstring.c -o cstring $ ./cstring
Ошибка
сегментации (сброшено ядро)
```

Упс! Первое, что нам нужно сделать, это выяснить, где произошел сбой сегмента. заперт. Тогда мы можем попытаться выяснить, почему это произошло.

Прежде чем продолжить, мы хотели бы упомянуть, что наш коллега по офису в соседнем отсеке, Милтон, также пытается исправить ошибки в этой программе. В отличие от нас, Милтон не знает, как использовать GDB, поэтому он собирается открыть Wordpad, вставить вызовы printf() по всему коду и перекомпилировать программу в попытке выяснить, где произошла ошибка сегмента. Давайте посмотрим, сможем ли мы отладить программу быстрее Милтона.

Пока Милтон открывает Wordpad, мы воспользуемся GDB для изучения основного файла:

```
$ gdb cstring core
Ядро было сгенерировано `cstring'.
Программа завершена сигналом 11, Ошибка сегментации.
#0 0x400a9295 в strcpy () из /lib/tls/libc.so.6

(gdb) обратная
трассировка #0 0x400a9295 в strcpy () из /lib/tls/libc.so.6 #1
0x080484df в Init_CString (str=0x80487c5 "Hello!") в cstring.c:15 #2 0x080485e4 в
main () в cstring.c:62
```

Согласно выходным данным обратной трассировки, ошибка сегмента произошла на строке 15, в `Init_CString()`, во время вызова `strncpy()`. Даже не глядя на код, мы уже знаем, что вероятность того, что мы передали `NULL`, довольно велика указатель на `strncpy()` в строке 15.

На данный момент Милтон все еще пытается решить, куда вставить первую часть много вызовов `printf()`.

4.3.1 Первый Ошибка

GDB сообщил нам, что ошибка сегмента произошла в строке 15 в `Init_CString()`, поэтому мы изменить текущий кадр на тот, который используется для вызова `Init_CString()`.

```
(gdb) кадр 1
#1 0x080484df в Init_CString (str=0x80487c5 "Привет!") в cstring.c:15
15         strncpy(p->str, str, strlen(str) + 1);
```

Мы применим принцип подтверждения, рассмотрев каждый из аргументов указателя, переданные в `strncpy()`, а именно, `str`, `p` и `p->str`, и проверка того, что их значения такие, какими мы считаем, они должны быть. Сначала мы печатаем значение `str`:

```
(gdb) печать стр
$1 = 0x80487c5 "Привет!"
```

Поскольку `str` является указателем, GDB выдает нам его значение в виде шестнадцатеричного адреса `0x80487c5`. А поскольку `str` — это указатель на `char`, а значит, адрес строки символов, GDB также любезно сообщает нам значение строки: «Привет!» Это также было ясно в выводе обратной трассировки, который мы видели выше, но мы должны проверить в любом случае. Итак, `str` не является `NULL` и указывает на допустимую строку, и пока все в порядке.

Теперь обратим внимание на другие аргументы указателя, `p` и `p->str`:

```
(gdb) печать *p
$2 = {
    стр = 0x0,
    длина = 6
}
```

Проблема теперь ясна: `p->str`, который также является указателем на строку, `NULL`. Это объясняет ошибку сегмента: мы попытались записать в ячейку памяти 0, которая нам недоступна.

Но что может привести к тому, что `p->str` (указатель строки в конструируемой `CString`) станет `NULL`? Ну, взглянем на код,

```
(gdb) список Init_CString
5     typedef структура {
6         символ *стр;
7         целая длина;
8     } CString;
```

```

9
10
11     CString *Init_CString(символ *строка)
12     {
13         CString *p = malloc(sizeof(CString));
14         p->len = strlen(str);
15         strncpy(p->str, str, strlen(str) + 1);
16         вернуть p;
17     }
18

```

мы видим, что перед строкой, в которой находится сегмент, находится всего две строки кода произошел сбой, и между ними, линия 13, скорее всего, является преступник.

Мы перезапустим программу из GDB, установим временную точку останова при входе в Init_CString() и пройдем по этой функции строкой за строкой, смотрим на значение p->str.

```
(gdb) tbreak Init_CString
Точка останова 1 по адресу 0x804849b: файл cstring.c, строка 13.
(gdb) запустить
```

```
Точка останова 1, Init_CString (str=0x80487c5 "Привет!") в cstring.c:13
13             CString *p = malloc(sizeof(CString));
(gdb) шаг
14                 p->len = strlen(str);
(gdb) печать p->str
$4 = 0x0
(gdb) шаг
15                 strncpy(p->str, str, strlen(str) + 1);
```

Вот в чем проблема: мы собираемся совершить ошибку сегмента, потому что следующий строка кода разыменовывает p->str, и p->str все еще NULL. Теперь мы используем маленький серые клеточки, чтобы выяснить, что произошло.

Когда мы выделили память для p, мы получили достаточно памяти для нашей структуры: указатель для хранения адреса строки и int для хранения длины строки, но мы не выделили память для хранения самой строки. Мы совершили распространенную ошибку, объявив указатель и не сделав его указывающим на что-либо! Что нам нужно сначала выделить достаточно памяти для хранения str, а затем сделать p->str указывают на эту недавно выделенную память. Вот как мы можем это сделать (мы нужно прибавить единицу к длине строки, потому что strlen() не учитывается завершающий '\0'):

```
CString *Init_CString(символ *строка)
{
    // Выделить для структуры
    CString *p = malloc(sizeof(CString));
    p->len = strlen(str);
```

```

// Выделить для строки
p->str = malloc(p->len + 1);
strncpy(p->str, str, strlen(str) + 1);
вернуть p;
}

```

Кстати, Милтон только что закончил добавлять вызовы printf() в свой код, и собирается перекомпилировать. Если ему повезет, он найдет, где ошибка сегмента произошла. Если нет, ему придется добавить еще больше вызовов printf() .

4.3.2 Не Оставлять ГБД В течение а Отладка Сессия

Во время сеанса отладки мы никогда не выходим из GDB, пока вносим изменения в наш код. Как обсуждалось ранее, таким образом мы избегаем длительных запусков, мы сохраняем наши точки останова и т. д.

Аналогично, мы оставляем открытый текстовый редактор. Оставаясь в том же редакторе сеанс между компиляциями во время отладки, мы можем с пользой использовать возможность «отмены» нашего редактора. Например, распространенная стратегия в процессе отладки — временное удаление частей кода, чтобы сосредоточиться на оставшиеся разделы, где, по вашему мнению, находится ошибка. После того, как вы закончите эту проверку, вы можете просто использовать функцию отмены редактора, чтобы восстановить удаленные строки.

Итак, на экране у нас обычно будет одно окно для GDB (или DDD) и одно окно для редактора. Мы также либо имели бы третье окно открытым для выдачи команд компилятору или, что еще лучше, выполнения их через редактор. Например, если вы используете редактор Vim, вы можете выполнить следующее команда, которая сохранит ваши изменения и перекомпилирует программу за один шаг:

:делать

(Мы предполагаем, что вы установили переменную автозаписи Vim , используя set autowrite, в файле запуска Vim. Эта функция Vim также переместит ваш курсор на первое сообщенное предупреждение или ошибку компиляции, если таковые имеются, и вы можно перемещаться вперед и назад по списку ошибок компиляции с помощью команд Vim :спехт и :сргев команды. Конечно, все это упрощается, если вы ставите короткие псевдонимы для этих команд в файле запуска Vim.)

4.3.3 Во-вторых и Третий Ошибки

После исправления первой ошибки мы снова запускаем программу из GDB. (помните, что когда GDB заметит, что вы перекомпилировали программу, он автоматически загрузит новый исполняемый файл, так что снова нет необходимости выходить и перезапустите GDB):

(gdb) запустить
Отлаживаемая программа уже запущена.
Начать с самого начала? (да или нет) да

'cstring' изменился; перечитываем символы.

Запуск программы: cstring

```
Инициализация:
str: «Привет!» len: 6
Chomp "":
    str:'Hello\0' len: 7 Append:
str: 'Hello!
0 world' len: 14
```

Программа завершилась нормально.

(gdb)

Похоже, что с Chomp() есть две проблемы . Во-первых, он должен был сократить '!', но, похоже, он сократил непечатаемый символ. Во-вторых, в конце нашей строки появляется нулевой символ. Поскольку Chomp() — очевидное место для поиска этих ошибок, мы запустим программу и поместим времененную точку останова на входе Chomp().

```
(gdb) tbreak Chomp
Breakpoint 2 at 0x8048523: файл cstring.c, строка 32. (gdb) run Запуск
программы:
cstring
```

```
Инициализация:
str: «Привет!» len: 6
```

```
Точка останова 1, Chomp (cstring=0x804a008) в cstring.c:32 char lastchar =
*(cstring->str + cstring->len); 32
(gdb)
```

Последний символ строки должен быть '!'. Давайте это проверим.

```
(gdb) вывести последний
символ $1 = 0 '\0'
```

Мы ожидали, что lastchar будет '!!', но вместо этого это нулевой символ. Похоже, это, вероятно, ошибка «off by one». Давайте разберемся. Мы можем визуализировать строку следующим образом:

```
смещение указателя: 0 1 2 3 4 5 6 cstring-
>str: Привет! \0 длина строки: 1 2 3 4 5 6
```

Последний символ строки хранится по адресу cstring->str + 5, но поскольку длина строки — это количество символов, а не индекс,

адрес `cstring->str + cstring->len` указывает на одну ячейку массива после последнего символа, где находится завершающий NULL , а не на то, куда мы хотели бы, чтобы он указывал. Мы можем исправить эту проблему, изменив строку 31 с

```
char lastchar = *( cstring->str + cstring->len);
```

К

```
char lastchar = *( cstring->str + cstring->len - 1);
```

В этой части кода скрывается третья ошибка. После вызова `Chomp()` строка «Hello!» стала «Hello!0» (вместо «Hello»). Следующая строка для выполнения в GDB, строка 33, — это то место, где мы хотели сократить строку, заменив ее последний символ на завершающий нулевой символ:

```
*( cstring->str + cstring->len) = '0';
```

Сразу же мы видим, что эта строка содержит ту же проблему, которую мы только что исправили в строке 31: мы ссылаемся на последний символ строки неправильно. Более того, теперь, когда наши глаза натренированы на этой строке кода, кажется, что мы сохраняем символ '0' в этом месте, который не является нулевым символом. Мы хотели поместить '0' в конец строки. После внесения этих двух исправлений строка 33 читается как

```
*( cstring->str + cstring->len - 1) = '\0';
```

На данный момент Милтон, наш коллега , использующий `printf()` , обнаружил ошибку первого сегмента и сейчас исправляет проблему выделения памяти в `Init_CString()`. Вместо того, чтобы заняться ошибками, которые мы только что исправили в `Chomp()`, ему придется заново удалить все вызовы `printf()` и перекомпилировать программу. Как неудобно!

4.3.4 Четвертый Ошибка

Давайте внесем исправления, обсуждавшиеся в предыдущем разделе, перекомпилируем код и снова запустим программу:

```
(gdb) run
'cstring' изменился; перечитывание символов.
Запуск программы: cstring
```

```
Инициализация:
str: «Привет!» len: 6
Chomp '!':
str: 'Привет' len: 5
Добавить:
str: 'Hello world' len: 12
```

Программа получила сигнал SIGSEGV, Ошибка сегментации.
0xb7f08da1 в свободной памяти () из /lib/tls/libc.so.6 (gdb)

Еще один сбой сегмента. Судя по отсутствующему восклицательному знаку после операция добавления, похоже, что следующая ошибка может скрываться в Append_Chars_To_CString(). Простая обратная трассировка должна подтвердить или опровергнуть эту гипотезу:

```
1 (gdb) обратная
трассировка 2 #0 0xb7f08da1 в free () из /lib/tls/libc.so.6 3 #1
0x0804851a в Delete_CString (p=0x804a008) в cstring3.c:24 4 #2 0x08048691 в main () в
cstring3.c:70 5 (gdb)
```

Согласно строке 3 вывода обратной трассировки, наше предположение неверно: программа на самом деле дала сбой в Delete_CString(). Это не значит, что у нас нет ошибки в Append_Chars_To_CString(), но наша непосредственная ошибка, та, которая вызвала ошибку seg, находится в Delete_CString(). Именно поэтому мы используем GDB здесь для проверки наших ожиданий — это полностью устраниет любые догадки при поиске того, где произошла ошибка seg. Как только наш друг , использующий printf(), доберется до этой точки в своей отладке, он поместит код трассировки в неправильную функцию!

К счастью, Delete_CString() короткий, поэтому мы сможем быстро найти ошибку.

```
(gdb) список Delete_CString
20
21 void Delete_CString(CString *p) 22 {
23     свободно(p);
24     свободно(p->str);
25 } 26
```

Сначала мы освобождаем p, затем освобождаем p->str. Это считается не такой уж и тонкой ошибкой. После освобождения p нет гарантии, что p->str больше не указывает на правильное место в памяти; оно может указывать куда угодно. В этом случае «где угодно» была память, к которой мы не могли получить доступ, что и привело к ошибке seg. Исправление заключается в том, чтобы изменить порядок вызовов free() на обратный:

```
void Delete_CString(CString *p) {

    свободно(p->str);
    свободно(p);
}
```

Кстати, Милтон слишком расстроился, пытаясь отследить отставание на единицу ошибка в Chomp(), которую мы так легко исправили. Это он звонит нам сейчас по телефону за помощью.

4.3.5 Пятый и Шестой Ошибки

Мы исправляем, перекомпилируем и перезапускаем код еще раз.

```
(gdb) run
`cstring' изменился; перечитывание символов.
Запуск программы: cstring
```

```
Инициализация:
str: «Привет!» len: 6
Chomp '!':
str:'Hello' len: 5 Append:
str: 'Hello
world' len: 12
```

Программа завершилась нормально.

```
(gdb)
```

После операции добавления мы пропустили восклицательный знак в строке, которая должна быть "Hello world!" Любопытно, что сообщенная длина строки 12 верна, хотя сама строка неверна. Наиболее логичным местом для поиска этой ошибки является Append_Chars_To_CString(), поэтому мы поместим точку останова там:

```
(gdb) tbreak Append_Chars_To_CString Точка
останова 3 по адресу 0x8048569: файл cstring.c, строка 45. (gdb) run
Запуск
программы: cstring
```

```
Инициализация:
ул: 'Привет!' длина: 6
Chomp '!':
str:'Hello' длина: 5
```

```
Точка останова 1, Append_Chars_To_CString (p=0x804a008, str=0x8048840 " мир!")
    в cstring.c:45 char
45          *newstr = malloc(p->len + 1);
```

Строка C newstr должна быть достаточно большой, чтобы вместить как p->str, так и str. Мы видим, что вызов malloc() в строке 45 не выделяет достаточно памяти; он выделяет достаточно места только для p->str и завершающего null. Строку 45 следует изменить на

```
char *newstr = malloc(p->len + strlen(str) + 1);
```

После внесения этого исправления и повторной компиляции мы получаем следующий результат:

```
(gdb) run
`cstring' изменился; перечитывание символов.
Запуск программы: cstring
```

```
Интерпретатор:
str: «Привет!» len: 6
Chomp '!':
str:'Hello' len: 5 Append:
str: 'Hello
world' len: 12
```

Наше исправление не исправило ошибку, которую мы имели в виду. То, что мы нашли и исправили, было «скрытой ошибкой». Не заблуждайтесь: это была ошибка, и тот факт, что она не проявилась как ошибка сегмента, был просто вопросом удачи. Вероятно, что оставшаяся ошибка все еще находится в Append_Chars_To_CString(), поэтому мы установим еще одну точку останова там:

```
(gdb) tbreak Append_Chars_To_CString Точка
останова 4 по адресу 0x8048569: файл cstring.c, строка 45. (gdb) run
Запуск
программы: cstring
```

```
(gdb) run
Init:
str: 'Привет!' len: 6 Chomp
'!': str:'Привет'
len: 5
```

```
Точка останова 1, Append_Chars_To_CString (p=0x804a008, str=0x8048840 " world!") в cstring.c:45 char
    *newstr = malloc(p-
45             >len + strlen(str) + 1);
(gdb) шаг 46
    p->len = p->len + strlen(str);
```

Строка 46 показывает, почему длина строки верна, хотя сама строка неверна: Сложение правильно вычисляет длину p->str, конкатенированной со str. Здесь нет проблем, поэтому мы сделаем шаг вперед.

```
(gdb) шаг
49         snprintf(newstr, p->len, "%s%s", p->str, str);
```

Следующая строка кода, строка 49, это то место, где мы формируем новую строку. Мы испытываем, что newstr будет содержать "Hello world!" после этого шага. Давайте применим принцип подтверждения и проверим это.

```
(gdb) шаг
51          free(p->str);
(gdb) print newstr $2
= 0x804a028 "Привет, мир"
```

В строке, построенной на строке 51, отсутствует восклицательный знак. Кода, поэтому ошибка, вероятно, возникает в строке 49, но что это может быть? В вызове snprintf() мы запросили, чтобы не более p->len байтов были скопированы в newstr. Значение p->len было подтверждено равным 12, а текст «Hello world!» содержит 12 символов. Мы не сказали snprintf() скопировать завершающий нулевой символ в исходной строке. Но тогда разве мы не должны были получить неправильно сформированную строку с восклицательным знаком в последней позиции и без нуля?

Это одна из замечательных особенностей snprintf(). Она всегда копирует завершающий нулевой символ в цель. Если вы ошибаетесь и укажете максимальное количество символов для копирования, которое меньше фактического количества символов в источнике (как мы сделали здесь), snprintf() скопирует столько символов, сколько сможет, но последний символ, записанный в цель, гарантированно будет нулевым символом. Чтобы исправить нашу ошибку, нам нужно сказать snprintf() скопировать достаточно байтов для хранения текста исходной строки и завершающего нулевого символа.

Поэтому нужно изменить строку 45. Вот полная, исправленная функция:

```
CString *Append_Chars_To_CString(CString *p, char *str) {

    char *newstr = malloc(p->len + strlen(str) + 1); p->len = p->len
    + strlen(str);

    // Создаем новую строку для замены p->str
    snprintf(newstr, p->len + 1, "%s%s", p->str, str);
    // Освобождаем старую строку и заставляем CString указывать на новую
    строку free(p-
    >str); p->str = newstr;

    вернуть p;
}
```

Давайте перекомпилируем исправленный код и запустим программу:

```
(gdb) run
`cstring' изменился; перечитывание символов.
Запуск программы: cstring
```

Интерпретатор:
str: `Привет!' len: 6 Чавкаю
!:

```
str:'Привет' длина: 5
```

Добавить:

```
str: 'Привет, мир!' длина: 12
```

Программа завершилась нормально.

(gdb)

Выглядит хорошо!

Мы охватили довольно большую территорию и столкнулись с некоторыми сложными концепциями, но это того стоило. Даже если наша глючная реализация CString была немного надуманной, наш сеанс отладки был довольно реалистичным и охватывал многие аспекты отладки:

- Принцип подтверждения •

Использование основных файлов для анализа процесса, который дал сбой •

Исправление, компиляция и повторный запуск программы без выхода из нее
ГБД

- Неадекватность отладки в стиле printf() • Использование

старой добродушной силы ума — этому нет замены

Если у вас есть опыт отладки в стиле printf(), вам может понадобиться подумайте, насколько сложнее было бы отслеживать некоторые из этих ошибок с помощью printf(). Диагностический код с использованием printf() имеет свое место в отладке, но как универсальный «инструмент» он совершенно неадекватен и неэффективен для отслеживания большинства ошибок, которые возникают в реальном коде.

5

ОТЛАДКА В КОНТЕКСТ МНОГОФУНКЦИОНАЛЬНОЙ ДЕЯТЕЛЬНОСТИ



Отладка — сложная задача, и она становится еще более сложной, когда некорректно работающее приложение пытается координировать несколько одновременных действий; клиент/сервер

Примерами этой парадигмы являются сетевое программирование, программирование с использованием потоков и параллельная обработка. В этой главе представлен обзор наиболее часто используемых методов мультипрограммирования и даны некоторые советы по устранению ошибок в таких программах, уделяя особое внимание использованию GDB/DDD/Eclipse в процессе отладки.

5.1 Отладка клиент-серверных сетевых программ

Компьютерные сети представляют собой чрезвычайно сложные системы, и тщательная отладка сетевых программных приложений иногда может потребовать использования аппаратных мониторов для сбора подробной информации о сетевом трафике.

Целая книга могла бы быть написана только по этой теме отладки. Наша цель здесь - чтобы просто представить тему.

Наш пример состоит из следующей пары клиент/сервер. Клиентское приложение позволяет пользователю проверить нагрузку на машину, на которой установлен сервер приложение запускается, даже если у пользователя нет учетной записи на последнем машина. Клиент отправляет запрос на информацию на сервер — здесь, запрос о загрузке на систему сервера, с помощью команды Unix `w` — через сетевое соединение. Затем сервер обрабатывает запрос и возвращает результаты, захват вывода `w` и отправка его обратно через соединение. В общем случае сервер может принимать запросы от нескольких удаленных клиентов; Чтобы упростить наш пример, предположим, что есть только один экземпляр клиента.

Код сервера показан ниже:

```

1 // srvr.c
2
3 // сервер для удаленного запуска команды w
4 // пользователь может проверить загрузку на машине без прав входа в систему
5 // использование: svr
6
7 #include <stdio.h>
8 #include <sys/types.h>
9 #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <netdb.h>
12 #include <fcntl.h>
13 #include <строка.h>
14 #include <unistd.h>
15 #include <stdlib.h>

16
17 #определить WPORT 2000
18 #define BUFSIZE 1000 // предполагается, что здесь достаточно
19
20 int clntdesc, // дескриптор сокета для отдельного клиента
21     svrdesc; // общий дескриптор сокета для сервера
22
23 символа outbuf[BUFSIZE]; // сообщения клиенту
24
25 недействительный ответ()
26     { int fd,nb;
27
28         memset(outbuf,0,sizeof(outbuf)); // очистить буфер
29         system("w > tmp.client"); // запускаем 'w' и сохраняем результаты
30         fd = open("tmp.client",O_RDONLY);
31         nb = read(fd,outbuf,BUFSIZE); // прочитать весь файл
32         write(clntdesc,outbuf,nb); // записываем его клиенту
33         unlink("tmp.client"); // удалить файл
34         закрыть(clntdesc);

```

```

35     }
36
37 целочисленный main()
38 { struct sockaddr_in bindinfo;
39
40     // создаем сокет, который будет использоваться для приема соединений
41     svrdesc = coket(AF_INET,SOCK_STREAM,0);
42     bindinfo.sin_family = AF_INET;
43     bindinfo.sin_port = WPORT;
44     bindinfo.sin_addr.s_addr = INADDR_ANY;
45     bind(svrdesc,(struct sockaddr *)&bindinfo,sizeof(bindinfo));
46
47     // OK, прослушиваем в цикле клиентские вызовы
48     слушать(svrdesc,5);
49
50     в то время как (1) {
51         // ждите звонка
52         clntdesc = принять(svrdesc,0,0);
53         // обработать команду
54         отвечать();
55     }
56 }
```

Вот код для клиента:

```

1 // clnt.c
2
3 // использование: clnt server_machine
4
5 #include <stdio.h>
6 #include <sys/types.h>
7 #включить <sys/socket.h>
8 #включить <netinet/in.h>
9 #включить <netdb.h>
10 #include <строка.h>
11 #включить <unistd.h>
12
13 #define WPORT 2000 // номер порта сервера
14 #define BUFSIZE 1000
15
16 int main(int argc, char **argv)
17 { int sd, размер_сообщения;
18
19     структура sockaddr_in адрес;
20     структура hostent *hostptr;
21     char buf[BUFSIZE];
22
23     // создать сокет
```

```

24     sd = сокет (AF_INET, SOCK_STREAM, 0);
25     addr.sin_family = AF_INET;
26     addr.sin_port = WPORT;
27     hostptr = gethostbyname(argv[1]);
28     memcpy(&addr.sin_addr.s_addr, hostptr->h_addr_list[0], hostptr->h_length);
29
30     // OK, теперь подключаемся
31     connect(sd,(struct sockaddr *) &addr,sizeof(addr));
32
33     // прочитать и отобразить ответ
34     msgsize = read(sd,buf,BUFSIZE);
35
36     если (размер_сообщения > 0)
37         запись(1,буфер,размер_сообщения);
38     printf("\n");
39     возврат 0;
}

```

Для тех, кто не знаком с клиент-серверным программированием, вот обзор как работают программы:

В строке 41 кода сервера вы создаете сокет, который является абстракцией, похожей на файловый дескриптор; точно так же, как файловый дескриптор используется для выполнения Операции ввода-вывода на объекте файловой системы, чтение и запись в сетевое соединение через сокет. В строке 45 сокет привязан к определенному

Номер порта, произвольно выбранный равным 2000. (Приложения пользовательского уровня, такие как (Этот номер ограничен номерами портов 1024 и выше.) Этот номер определяет «почтовый ящик» в системе сервера, на который клиенты отправляют запросы быть обработаны для данного конкретного приложения.

Сервер «открывается для бизнеса» путем вызова listen() в строке 48. Затем он ожидает поступления клиентского запроса, вызывая accept() в строке 52. Этот вызов блокируется до тех пор, пока не поступит запрос. Затем он возвращает новый сокет для связи с клиентом. (Когда есть несколько клиентов, исходный сокет продолжает принимать новые запросы, даже если существующий запрос обслуживается, отсюда и потребность в отдельных сокетах. Это потребовало бы от сервера быть реализовано в потоковом режиме.) Сервер обрабатывает клиентский запрос с помощью функции response() и отправляет информацию о загрузке машины клиент, локально вызвав команду w и записав результаты в разъем в строке 32.

Клиент создает сокет на строке 24, а затем использует его на строке 31 для подключения к порту 2000 сервера. На строке 34 он считывает отправленную информацию о нагрузке сервером, а затем распечатывает его.

Вот как должен выглядеть результат работы клиента:

```
$ clnt laura.cs.ucdavis.edu
13:00:15 до 13 дней, 39 мин, 7 пользователей, средняя нагрузка: 0,25, 0,13, 0,09
ПОЛЬЗОВАТЕЛЬ ИЗ LOGIN@ IDLE JCPU PCPU ЧТО
матлофф :0          -          14июн07 ?xdm? 25:38 0.15c /bin/tcsh -c /
матлофф баллов/1    :0.0      14июн07 17:34 0.46c 0.46c -csh
матлофф баллов/2    :0.0      14июн07 18:12 0.39c 0.39c -csh
```

матлофф баллов/3	:0.0	14 июня 07 г. 58,00 с. 2,18 с. 2,01 с. /usr/bin/mutt
матлофф баллов/4	:0.0	14июня07 0.00с 1.85с 0.00с clnt laura.cs.u
матлофф баллов/5	:0.0	14июн07 20.00с 1.88с 0.02с скрипт
матлофф баллов/7	:0.0	19июн07 4дня 22:17 0.16с -csh

Теперь предположим, что программист забыл строку 26 в клиентском коде, который указывает порт в системе сервера для подключения:

```
addr.sin_port = WPORT;
```

Давайте притворимся, что мы не знаем, в чем заключается ошибка, и посмотрим, как мы можем ее отследить. Это вниз.

Выходные данные клиента теперь будут такими:

```
$ clnt laura.cs.ucdavis.edu
```

```
$
```

Судя по всему, клиент вообще ничего не получил от сервера.

Конечно, это может быть вызвано различными причинами как на сервере, так и на клиент или оба.

Давайте осмотримся, используя GDB. Сначала проверьте, что клиент на самом деле удалось подключиться к серверу. Установите точку останова на вызове для connect() и запуска программы:

```
(гdb) б 31
```

Точка останова 1 по адресу 0x8048502: файл clnt.c, строка 31.

```
(gdb) r laura.cs.ucdavis.edu
```

Запуск программы: /fandrhome/matloff/public_html/matloff/public_html/Debug
/Книга/DDD/clnt laura.cs.ucdavis.edu

Точка останова 1, основная (argc=2, argv=0xbff81a344) в clnt.c:31

```
31      connect(sd,(struct sockaddr *)&addr,sizeof(addr));
```

Используйте GDB для выполнения connect() и проверки возвращаемого значения на наличие ошибки:

```
(gdb) p connect(sd,&addr,sizeof(addr))
```

```
$1 = -1
```

Это действительно -1, код неудачи. Это большой намек. (Конечно, как вопрос защитного программирования, когда мы писали клиентский код, мы бы проверили возвращаемое значение connect() и обработали случай сбоя соединять.)

Кстати, обратите внимание, что при ручном выполнении вызова connect() вы придется снять гипс. Если гипс останется, вы получите ошибку:

```
(gdb) p connect(sd,(struct sockaddr *) &addr,sizeof(addr))
```

Нет типа структуры с именем sockaddr.

Это связано с особенностью GDB, и она возникает из-за того, что мы не использовали struct в другом месте программы.

Также обратите внимание, что если бы попытка connect() была успешной в сеансе GDB, вы бы не смогли продолжить и выполнить строку 31. Попытка открыть уже открытый сокет является ошибкой.

Вам пришлось бы пропустить строку 31 и перейти сразу к строке 34. Вы могли бы сделать это с помощью команды jump GDB , выполнив jump 34, но в целом вам следует использовать эту команду с осторожностью, так как это может привести к пропуску некоторых машинных инструкций, которые необходимы далее в коде. Поэтому, если попытка соединения была успешной, вам, вероятно, захочется перезапустить программу.

Давайте попробуем выяснить причину неудачи, проверив аргумент addr в вызове connect():

```
(gdb) p адрес
...
connect(3, {sa_family=AF_INET, sin_port=htons(1032),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 ECONNREFUSED (В соединении отказано)
...
```

Ага! Значение htons(1032) указывает на порт 2052 (см. ниже), а не 2000 мы ожидаем. Это говорит о том, что вы либо неправильно указали порт, либо вообще забыли его указать. Если вы проверите, то быстро обнаружите, что последнее имело место.

Опять же, было бы разумно включить немного машин в исходный код, чтобы помочь процессу отладки, например, проверку возвращаемых значений системных вызовов. Другим полезным шагом является включение строки

```
#include <errno.h>
```

что в нашей системе создает глобальную переменную errno, значение которой можно вывести из кода или из GDB:

```
(gdb) для
ошибки $1 = 111
```

В файле /usr/include/linux/errno.h вы обнаружите, что этот номер ошибки кодирует ошибку отказа в подключении .

Однако реализация библиотеки errno может отличаться от платформы к платформе. Например, файл заголовка может иметь другое имя или errno может быть реализован как вызов макроса вместо переменной.

Другой подход — использовать strace, который отслеживает все системные вызовы, выполняемые программой:

```
$ strace clnt laura.cs
...
connect(3, {sa_family=AF_INET, sin_port=htons(1032),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 ECONNREFUSED (В соединении отказано)
...
```

Это дает вам две важные части информации по цене одной. Во-первых, вы сразу видите, что произошла ошибка ECONNREFUSED . Во-вторых, вы также видите, что порт был htons(1032), который имеет значение 2052. Вы можете проверить это последнее значение, выполнив команду типа

```
(gdb) p htons(1032)
```

из GDB, который показывает значение 2052, что, очевидно, не равно 2000, как ожидалось.

Вы обнаружите, что strace — удобный инструмент во многих контекстах (сетевых и в противном случае) для проверки результатов системных вызовов.

В качестве другого примера предположим, что вы случайно пропустили запись в клиент в коде сервера (строка 32):

```
write(clntdesc,outbuf,nb); // записываем его клиенту
```

В этом случае клиентская программа зависнет, ожидая ответа, который не поступит. Конечно, в этом простом примере вы сразу же заподозрите проблему с вызовом write() на сервере и быстро обнаружите, что мы забыли об этом. Но в более сложных программах причина может быть не столь очевидной. В таких случаях вы, вероятно, настроите два одновременных сеанса GDB, один для клиента и один для сервера, последовательно проходя через обе программы. Вы обнаружите, что в какой-то момент их совместной работы клиент зависает, ожидая ответа от сервера, и таким образом получите подсказку о вероятном местоположении ошибки на сервере. Затем вы сосредоточите свое внимание на сеансе GDB сервера, пытаясь выяснить, почему он не отправил клиенту в этот момент.

В действительно сложных случаях отладки сети можно использовать программу Ethereal с открытым исходным кодом для отслеживания отдельных пакетов TCP/IP.

5.2 Отладка потокового кода

Поточное программирование стало довольно популярным. Для Unix наиболее распространенным пакетом потоков является стандарт POSIX Pthreads, поэтому мы будем использовать его для нашего примера в этом разделе. Принципы аналогичны для других пакетов потоков.

5.2.1 Обзор изПроцессыи Темы

Современные операционные системы используют разделение времени для управления несколькими запущенными программами таким образом, что пользователю кажется, что они выполняются одновременно.

Конечно, если машина имеет более одного процессора, более одной программы на самом деле могут работать одновременно, но для простоты мы предположим только один процессор, в этом случае одновременность лишь кажущаяся.

Каждый экземпляра запущенной программы представляется ОС как процесс (в терминологии Unix) или задача (в Windows). Таким образом, множественные вызовы одна программа, которая выполняется одновременно (например, одновременные сеансы) текстового редактора vi) — это отдельные процессы. Процессы должны «поочередно» на машине с одним процессором. Для конкретности предположим, что «повороты», называемые временными интервалами, имеют длину 30 миллисекунд.

После того, как процесс проработает 30 миллисекунд, аппаратный таймер выдает сигнал прерывания, которое заставляет ОС работать. Мы говорим, что процесс был прерван. ОС сохраняет текущее состояние прерванного процесса, чтобы он мог быть возобновлено позже, затем выбирает следующий процесс, чтобы дать ему временной интервал. Это называется переключением контекста, поскольку среда выполнения ЦП имеет переключился с одного процесса на другой. Этот цикл повторяется бесконечно.

Ход может закончиться раньше времени. Например, когда процессу необходимо выполнить ввод/вывод, он в конечном итоге вызывает функцию в ОС, которая выполняет низкоуровневые аппаратные операции; например, вызов функции библиотеки C `scanf()` приводит к вызову системного вызова `read()` ОС Unix, который взаимодействует с драйвером клавиатуры. Таким образом, процесс уступает свою очередь ОС, и ход заканчивается рано.

Одним из следствий этого является то, что планирование временных интервалов для данного процесса довольно случайно. Время, которое требуется пользователю, чтобы подумать и затем нажать key является случайным, поэтому время начала его следующего временного среза непредсказуемо. Более того, если вы отлаживаете потоковую программу, вы не знаете порядок, в котором будут запланированы потоки; это может сделать отладку более трудный.

Вот немного больше подробностей: ОС поддерживает таблицу процессов, в которой перечислена информация обо всех текущих процессах. Грубо говоря, каждый процесс отмечены в таблице как находящиеся либо в состоянии Run, либо в состоянии Sleep. Давайте рассмотрим пример, в котором запущенная программа достигает точки, в которой она необходимо прочитать ввод с клавиатуры. Как только что было отмечено, это завершит ход процесса. Поскольку процесс теперь ждет завершения ввода-вывода, ОС помечает его как находящийся в состоянии сна, что делает его непригодным для использования временных интервалов. Таким образом, нахождение в состоянии сна означает, что процесс заблокирован и ожидает некоторое событие должно произойти. Когда это событие наконец произойдет позже, ОС будет затем измените его состояние в таблице процессов обратно на «Выполнить».

События, не связанные с вводом-выводом, также могут вызывать переход в состояние сна. Например, если родительский процесс создает дочерний процесс и вызывает `wait()`, родительский процесс будет заблокирован до тех пор, пока дочерний процесс не завершит свою работу и не завершится. Опять же, точно Когда это происходит, обычно непредсказуемо.

Более того, нахождение в состоянии «Выполнение» не означает, что процесс фактически выполняется на ЦП; скорее, это просто означает, что он готов запустить — то есть, иметь право на квант времени процессора. При переключении контекста, ОС выбирает процесс, которому будет предоставлена следующая возможность запустить ЦП, из числа которые в настоящее время находятся в состоянии Run, согласно таблице процессов. Процедура планирования, используемая ОС для выбора нового контекста, гарантирует, что любой

данный процесс будет продолжать получать временные срезы и в конечном итоге завершится, но нет никаких обещаний относительно того, какие временные срезы он получит. Таким образом, точное время, когда спящий процесс фактически «пробуждается» после того, как произошло событие, которого он ожидает, является случайным, как и точная скорость прогресса процесса к завершению.

Поток во многом похож на процесс, за исключением того, что он разработан так, чтобы занимать меньше памяти и тратить меньше времени на создание и переключение между ними, чем процессы. Действительно, потоки иногда называют «легкими» процессами, и в зависимости от системы потоков и среды выполнения они могут быть даже реализованы как процессы операционной системы. Подобно программам, которые порождают процессы для выполнения работы, многопоточное приложение обычно выполняет процедуру `main()`, которая создает один или несколько дочерних потоков. Родительский поток, `main()`, также является потоком.

Основное различие между процессами и потоками заключается в том, что, хотя каждый поток имеет свои собственные локальные переменные, как и в случае с процессом, глобальные переменные родительской программы в потоковой среде являются общими для всех потоков и служат основным методом связи между потоками. (Возможно совместное использование глобальных переменных между процессами Unix, но это неудобно.)

В системе Linux вы можете просмотреть все процессы и потоки, запущенные в данный момент в системе, выполнив команду `ps auxH`.

Существуют системы невытесняющих потоков, но Pthreads использует вытесняющую политику управления потоками, и поток в программе может быть прерван в любой момент другим потоком. Таким образом, элемент случайности, описанный выше для процессов в системе с разделением времени, также возникает в поведении потоковой программы. В результате некоторые ошибки в приложениях, разработанных с использованием Pthreads, нелегко воспроизвести.

5.2.2 Базовый Пример

Мы не будем усложнять и воспользуемся следующим кодом для поиска простых чисел в качестве примера. Программа использует классическое решето Эратосфена. Чтобы найти все простые числа от 2 до n , мы сначала перечисляем все числа, затем вычеркиваем все кратные 2, затем все кратные 3 и так далее. Все оставшиеся в конце числа являются простыми числами.

```

1 // находит простые числа от 2 до n; использует решето Эратосфена, 2 // удаляет все числа,
кратные 2, все числа, кратные 3, все числа, кратные 5, 3 // и т. д.; неэффективно, например, каждый
поток должен удалять целый блок из 4 // значений base перед тем, как перейти к nextbase за
другими
5
6 // использование: sieve nthreads n 7 //
где nthreads — количество рабочих потоков
8
9 #include <stdio.h>
10 #include <math.h> 11
11 #include <pthread.h>
12
13 #определить MAX_N 100000000

```

```

14 #определить МАКС_ПОТОКОВ 100
15
16 // общие переменные

17 int nthreads, // количество потоков (не считая main())
18     n, // верхняя граница диапазона, в котором нужно найти простые числа
19     prime[MAX_N+1], // в конце концов, prime[i] = 1, если i простое, иначе 0
20     nextbase; // следующий множитель решета, который будет использоваться
21
22 int work[MAX_THREADS]; // чтобы измерить объем работы, выполняемой каждым потоком,
23                         // по количеству проверенных множителей решета
24
25 // блокировка индекса для общей переменной nextbase
26 pthread_mutex_t nextbaselock = PTHREAD_MUTEX_INITIALIZER;
27
28 // Структуры идентификаторов для потоков
29 pthread_t id[MAX_THREADS];
30
31 // "вычеркивает" все кратные k, начиная с k*k
32 пустое пересечение (целое k)
33 { int i;
34
35     для (я = k; я*k <= n; я++) {
36         простое число[i*k] = 0;
37     }
38 }
39
40 // процедура рабочего потока
41 void *worker(int tn) // tn — номер потока (0,1,...)
42 { int lim,base;
43
44     // нет необходимости проверять множители больше sqrt(n)
45     лим = кв. корень(n);
46
47     делать {
48         // получаем следующий множитель решета, избегая дублирования между потоками
49         pthread_mutex_lock(&nextbaselock);
50         база = следующаябаза += 2;
51         pthread_mutex_unlock(&nextbaselock);
52         если (база <= предел) {
53             work[tn]++;
54             // не беспокойтесь о вычеркивании, если известно, что база
55             // композитный
56             если (простое[основание])
57                 кроссаут(база);
58         }
59         в противном случае возврат;
60     } пока (1);

```

```

61    }
62
63 main(целое число argc, символ **argv)
64     { int nprimes, // количество найденных простых чисел
65      totwork, // количество проверенных базовых значений
66      я;
67      недействительный *р;
68
69      n = atoi(argv[1]);
70      nthreads = atoi(argv[2]);
71      для (i = 2; i <= n; i++)
72          простое число[i] = 1;
73      вычеркнуть(2);
74      следующаябаза = 1;
75      // начать потоки
76      для (i = 0; i < nthreads; i++) {
77          pthread_create(&id[i],NULL,(void *) worker,(void *) i);
78      }
79
80      // ждем пока все будет сделано
81      работа = 0;
82      для (i = 0; i < nthreads; i++) {
83          pthread_join(id[i],&p);
84          printf("%d значений базы выполнено\n",work[i]);
85          totwork += работа[i];
86      }
87      printf("%d всего значений базы выполнено\n",totwork);
88
89      // отчет о результатах
90      nпростых чисел = 0;
91      для (i = 2; i <= n; i++)
92          если (простое число[i]) nпростых чисел++;
93      printf("количество найденных простых чисел составило %d\n",nprimes);
94
95 }

```

В этой программе есть два аргумента командной строки, верхний ограничен по диапазону, который необходимо проверить на наличие простых чисел, и nthreads, число рабочих потоков, которые мы хотим создать.

Здесь main() создает рабочие потоки, каждый из которых является вызовом функции worker(). Рабочие разделяют три элемента данных: верхний связанный переменная n; переменная, указывающая следующее число, кратные которого должны быть исключены из диапазона 2..n, nextbase; и массив prime[] который записывает для каждого числа в диапазоне 2..n, было ли оно исключено. Каждый вызов многократно извлекает еще не обработанный множитель исключения, основание, а затем исключает все кратные основания из диапазон 2..n. После создания рабочих процессов main() использует pthread_join() для ожидания

все эти потоки, чтобы закончить свою работу перед возобновлением работы, в этот момент он подсчитывает оставшиеся простые числа и выдает свой отчет. Отчет включает в себя не только количество простых чисел, но и информацию о том, сколько работы выполнил каждый рабочий поток. Эта оценка полезна для балансировки нагрузки и оптимизации производительности в многопроцессорной системе.

Каждый экземпляр `worker()` извлекает следующее значение `base`, выполняя следующий код (строки 49–51):

```
pthread_mutex_lock(&nextbaselock); база =
nextbase += 2;
pthread_mutex_unlock(&nextbaselock);
```

Здесь глобальная переменная `nextbase` обновляется и используется для инициализации значения локальной переменной `base` экземпляра `worker()`; затем `worker` вычеркивает кратные `base` в массиве `prime[]`. (Обратите внимание, что мы начали с исключения всех кратных 2 в начале `main()`, и в дальнейшем нам нужно учитывать только нечетные значения для `base`.)

Как только рабочий поток узнает значение `base` для использования, он может безопасно вычеркнуть кратные `base` из общего массива `prime[]`, поскольку никакой другой рабочий поток не будет использовать это значение `base`. Однако нам нужно разместить защитные операторы вокруг операции обновления общей переменной `nextbase`, от которой зависит `base` (строка 26). Напомним, что любой рабочий поток может быть вытеснен в непредсказуемое время другим рабочим потоком, который окажется в непредсказуемом месте в коде для `worker()`. В частности, может случиться так, что текущий рабочий поток будет прерван в середине оператора

```
база = следующаябаза += 2;
```

и следующий временной срез отдается другому потоку, который также выполняет тот же оператор. В этом случае два рабочих одновременно пытаются изменить общую переменную `nextbase`, что может привести к коварным и трудновоспроизводимым ошибкам.

Заключение кода, который манипулирует общей переменной, известной как критическая секция, в защитные операторы предотвращает возникновение этой ситуации. Вызовы `pthread_mutex_lock()` и `pthread_mutex_unlock()` гарантируют, что существует не более одного потока, выполняющего заключенный в нем фрагмент программы. Они сообщают ОС, что нужно разрешить потоку войти в критическую секцию, только если в данный момент нет другого потока, выполняющего ее, и не вытеснять этот поток, пока он не завершит всю секцию. (Переменная блокировки `nextbaselock` используется внутри системы потоков для обеспечения этого «взаимного исключения».)

К сожалению, слишком легко не распознать и/или неправильно спрогнозировать течь критические разделы в потоковом коде. Давайте посмотрим, как GDB может быть использован для отладки такого рода ошибок в программе Pthreads. Предположим, мы забыли включить оператор разблокировки,

```
pthread_mutex_unlock(&nextbaselock);
```

Это, конечно, приводит к зависанию программы, как только в критическую секцию впервые войдет рабочий поток, поскольку другие рабочие потоки будут ждать вечно, пока блокировка не будет снята. Но давайте сделаем вид, что вы этого еще не знаете. Как вы отследите виновника с помощью GDB?

Скомпилируйте программу, убедившись, что включены флаги `-fthread -lm` для присоединения Pthreads и математических библиотек (последняя необходима для вызова `sqrt()`). Затем запустите код в GDB с `n = 100` и `nthreads = 2`:

```
(гdb) г 100 2
Запуск программы: /debug/primes 100 2
[Новая тема 16384 (LWP 28653)]
[Новая тема 32769 (LWP 28676)]
[Новая тема 16386 (LWP 28677)]
[Новая тема 32771 (LWP 28678)]
```

Каждый раз, когда создается новый поток, GDB объявляет об этом, как показано здесь. Мы рассмотрим, что это за поток, чуть позже.

Программа зависает, и вы прерываете ее нажатием CTRL-C. GDB Теперь сессия выглядит так:

```
(гdb) г 100 2
Запуск программы: /debug/primes 100 2
[Новая тема 16384 (LWP 28653)]
[Новая тема 32769 (LWP 28676)]
[Новая тема 16386 (LWP 28677)]
[Новая тема 32771 (LWP 28678)]
```

Программа получила сигнал SIGINT, прерывание.
[Переключение на поток 32771 (LWP 28678)]
0x4005ba35 в `_pthread_sigsuspend ()` из `/lib/i686/libpthread.so.0`

В такой ситуации крайне важно знать, что делает каждый поток. Это можно определить с помощью команды GDB `info threads`:

```
(gdb) информация о
потоках * 4 Поток 32771 (LWP 28678) 0x4005ba35 в _pthread_sigsuspend () из /lib/
i686/libpthread.so.0 3 Поток 16386
(LWP 28677) 0x4005ba35 в _pthread_sigsuspend () из /lib/i686/libpthread.so.0 2
Поток 32769 (LWP 28676) 0x420db1a7
в poll () из
/lib/i686/libc.so.6
1 Поток 16384 (LWP 28653) 0x4005ba35 в _pthread_sigsuspend () из /lib/i686/
libpthread.so.0
```

Звездочка означает, что вы сейчас находитесь в теме 4. Давайте посмотрим, что продолжая эту тему:

```
(gdb) bt
#0 0x4005ba35 в __pthread_sigsuspend () из /lib/i686/libpthread.so.0 #1
0x4005adb8 в __pthread_wait_for_restart_signal ()
    из /lib/i686/libpthread.so.0 #2
0x4005d190 в __pthread_alt_lock () из /lib/i686/libpthread.so.0 #3 0x40059d77
в pthread_mutex_lock () из /lib/i686/libpthread.so.0 #4 0x0804855f в worker
(tn=1) в Primes.c:49 #5 0x40059881 в
pthread_start_thread () из /lib/i686/libpthread.so.0 #6 0x40059985 в
pthread_start_thread_event () из /lib/i686/libpthread.so.0
```

(Это работает в реализации Pthreads в LinuxThreads, но может быть невозможно на некоторых других платформах.)

Ага, вы видите на кадрах 3 и 4, что эта ветка находится на строке 49 источника. код и пытается получить блокировку и войти в критическую секцию:

```
pthread_mutex_lock(&nextbaselock);
```

Обратите внимание также на кадр 0 выше, что поток, по-видимому, приостановлен в ожидании снятия блокировки другим потоком. Он не получит никаких временных срезов, пока это не произойдет и менеджер потоков не организует получение блокировки.

Что делают другие потоки? Вы можете проверить стек любого потока, переключившись на этот поток и выполнив команду `bt`:

```
(gdb) поток 3
[Переключение на поток 3 (поток 16386 (LWP 28677))]:#0 0x4005ba35 в
__pthread_sigsuspend () из /lib/i686/libpthread.so.0 (gdb)
bt #0
0x4005ba35 в __pthread_sigsuspend () из /lib/i686/libpthread.so.0 #1
0x4005adb8 в __pthread_wait_for_restart_signal ()
    из /lib/i686/libpthread.so.0 #2
0x4005d190 в __pthread_alt_lock () из /lib/i686/libpthread.so.0 #3 0x40059d77
в pthread_mutex_lock () из /lib/i686/libpthread.so.0 #4 0x0804855f в worker
(tn=0) в Primes.c:49 #5 0x40059881 в
pthread_start_thread () из /lib/i686/libpthread.so.0 #6 0x40059985 в
pthread_start_thread_event () из /lib/i686/libpthread.so.0
```

Вспомните, что мы создали два рабочих потока. Выше вы видели, что поток 4 был одним из них (кадр 4 из его вывода `bt`), а теперь вы видите из кадра 4 вывода здесь, что поток 3 является другим. Вы также видите, что поток 3 также пытается получить блокировку (кадр 3).

Других рабочих потоков быть не должно, но один из основных принципов отладки заключается в том, что ничто не принимается на веру, и все должно быть проверено. Мы делаем это сейчас, проверяя состояние оставшихся

потоки. Вы обнаружите, что два других потока являются нерабочими потоками, как следует:

```
(gdb) поток 2
[Переключение на поток 2 (поток 32769 (LWP 28676))]#0 0x420db1a7 в опросе
()
из /lib/i686/libc.so.6
(гдб) бт
#0 0x420db1a7 в опросе () из /lib/i686/libc.so.6
#1 0x400589de в __pthread_manager () из /lib/i686/libpthread.so.0
#2 0x4005962b в __pthread_manager_event () из
/lib/i686/libpthread.so.0
```

Итак, поток 2 — это менеджер потоков. Это внутреннее для Pthreads package. Это определенно не рабочий поток, что частично подтверждает наши ожидания, что рабочих потоков всего два. Проверка потока 1,

```
(gdb) поток 1
[Переключение на поток 1 (поток 16384 (LWP 28653))]#0 0x4005ba35 в
__pthread_sigsuspend() из /lib/i686/libpthread.so.0
(гдб) бт
#0 0x4005ba35 в __pthread_sigsuspend() из /lib/i686/libpthread.so.0
#1 0x4005adb8 в __pthread_wait_for_restart_signal ()
из /lib/i686/libpthread.so.0
#2 0x40058551 в pthread_join () из /lib/i686/libpthread.so.0
#3 0x080486aa в main (argc=3, argv=0xbffffe7b4) в Primes.c:83
#4 0x420158f7 в __libc_start_main () из /lib/i686/libc.so.6
```

вы обнаружите, что он выполняет main() и таким образом можете подтвердить, что есть только два рабочие потоки.

Однако оба рабочих застряли, ожидая, пока замок закроется. быть отброшен. Неудивительно, что программа зависла! Этого достаточно, чтобы точно определить место и природу ошибки, и мы быстро понимаем, что забыли вызвать функцию разблокировки.

5.2.3 А Вариация

Что, если бы вы не осознали необходимость защиты обновления общего доступа? переменная nextbase в первую очередь? Что бы произошло в предыдущем примере, если бы вы пропустили и разблокировку, и блокировку?

Наивный взгляд на этот вопрос может привести к предположению, что не было никакого вреда с точки зрения правильной работы программы (т. е. получения точного подсчета количества простых чисел), хотя, возможно, и с замедлением из-за дублирования работы (т. е. использования одного и того же значения основания более один раз). Кажется, что некоторые потоки могут дублировать работу других, а именно, когда два рабочих случайно захватывают одно и то же значение nextbase для инициализации своих локальных копий base. Некоторые составные числа могут затем закончиться

если бы число было зачеркнуто дважды, то результаты (т. е. подсчет количества простых чисел) все равно были бы правильными.

Но давайте взглянем поближе. Заявление

база = следующая база += 2;

компилируется как минимум в две инструкции машинного языка. Например, при использовании компилятора GCC на машине Pentium под управлением Linux приведенный выше оператор C преобразуется в следующие инструкции языка ассемблера (полученные путем запуска GCC с опцией `-S` и последующего просмотра полученного файла `.s`):

добавить \$2, следующая база
переместить следующая база,
%eax переместить %eax, -8(%ebp)

Этот код увеличивает nextbase на 2, затем копирует значение nextbase в регистр EAX и, наконец, копирует значение EAX в то место в стеке рабочего процесса, где хранится его локальная переменная `base`.

Предположим, у вас есть только два рабочих потока, а значение nextbase равно, скажем, 9, а текущий временной интервал вызова `worker()` заканчивается сразу после того, как он выполнит машинную инструкцию.

добавить \$2, следующая база

который устанавливает общую глобальную переменную `nextbase` в 11. Предположим, что следующий временной отрезок переходит к другому вызову `worker()`, который выполняет те же самые инструкции. Второй `worker` теперь увеличивает `nextbase` до 13, использует это для установки своей локальной переменной `base` и начинает исключать все кратные 13. В конце концов, первый вызов `worker()` получит еще один временной отрезок, и затем он продолжит с того места, где остановился, выполняя машинные инструкции

`movl nextbase, %eax movl
%eax, -8(%ebp)`

Конечно, значение `nextbase` теперь равно 13. Таким образом, первый рабочий устанавливает значение своей локальной переменной `base` равным 13 и приступает к исключению кратных этому значению, а не значению 11, которое он установил во время своего последнего временного среза. Ни один рабочий не делает ничего с кратными 11. В итоге вы не только дублируете работу без необходимости, но и пропускаете необходимую работу!

Как можно обнаружить такую ошибку с помощью GDB? Предположительно «симптомом», который всплыл, заключался в том, что количество сообщенных простых чисел было слишком большим. Таким образом, вы можете заподозрить, что значения базы иногда каким-то образом пропускаются. Чтобы проверить эту гипотезу, вы можете разместить точку останова сразу после строки

база = следующая база += 2;

Повторно выполнив команду GDB `continue (c)` и отобразив значение `base`,

(gdb) база дисп

В конечном итоге вы можете убедиться, что значение `base` действительно пропущено.

Ключевое слово здесь — может. Вспомните наше предыдущее обсуждение, что потоковые программы работают в несколько случайной манере. В контексте здесь это может быть случай, когда при некоторых запусках программы ошибка всплывает (т.е. слишком много простые числа сообщаются), но в других запусках вы можете получить правильные ответы!!

К сожалению, хорошего решения этой проблемы не существует. Отладка многопоточный код часто требует дополнительного терпения и креативности.

5.2.4 ГБД Темы Команда

[Краткое содержание](#)

Ниже приведено краткое описание использования команд GDB, связанных с потоками:

- информационные потоки (дает информацию обо всех текущих потоках)
- поток 3 (Изменения в потоке 3)
- break 88 thread 3 (Останавливает выполнение, когда поток 3 достигает исходной строки 88)
- прервать 88 поток 3, если `x==y` (Останавливает выполнение, когда поток 3 достигает источника строки 88 и переменные `x` и `y` равны)

5.2.5 Темы Команды в DDD

В DDD выберите Статус | Темы, и появится окно, отображающее все потоки в стиле информационных потоков GDB, как показано на рисунке 5-1. Вы можете щелкнуть по потоку, чтобы переключить на него фокус отладчика.

Вы, вероятно, захотите оставить это всплывающее окно, а не используя его один раз и затем закрывая его. Таким образом, вам не придется снова открывать каждый раз, когда вы хотите увидеть, какой поток в данный момент запущен или переключиться на другую тему.

Похоже, нет способа сделать точку останова специфичной для потока в DDD, как вы сделали с командой GDB `break 88 thread 3` выше. Вместо этого, Вы подаете такую команду GDB через консоль DDD.

5.2.6 Темы Команды в Eclipse

Обратите внимание, что make-файл по умолчанию, созданный Eclipse, не будет включать `-lpthread` аргумент командной строки для GCC (он также не будет включать аргументы для любых других специальных библиотек, которые вам нужны). Вы можете изменить `makefile` напрямую, если хотите, но проще сказать Eclipse сделать это за вас. Пока в перспективе C/C++, щелкните правой кнопкой мыши имя вашего проекта и выберите «Свойства»; направьте треугольник рядом с C/C++ Build вниз; выберите «Настройки» | «Инструмент» Настройки; наведите треугольник рядом с GCC C Linker вниз и выберите Li-braries | Add (последний — это зеленый значок плюса); и заполните свою библиотеку флаги минус `-l` (например, заполнение `pt` вместо `-lpt`). Затем соберите свой проект.

Вспомните главу 1 , в которой Eclipse постоянно отображает список ваших тем. в отличие от необходимости запрашивать его, как в DDD. Более того, вам не нужно

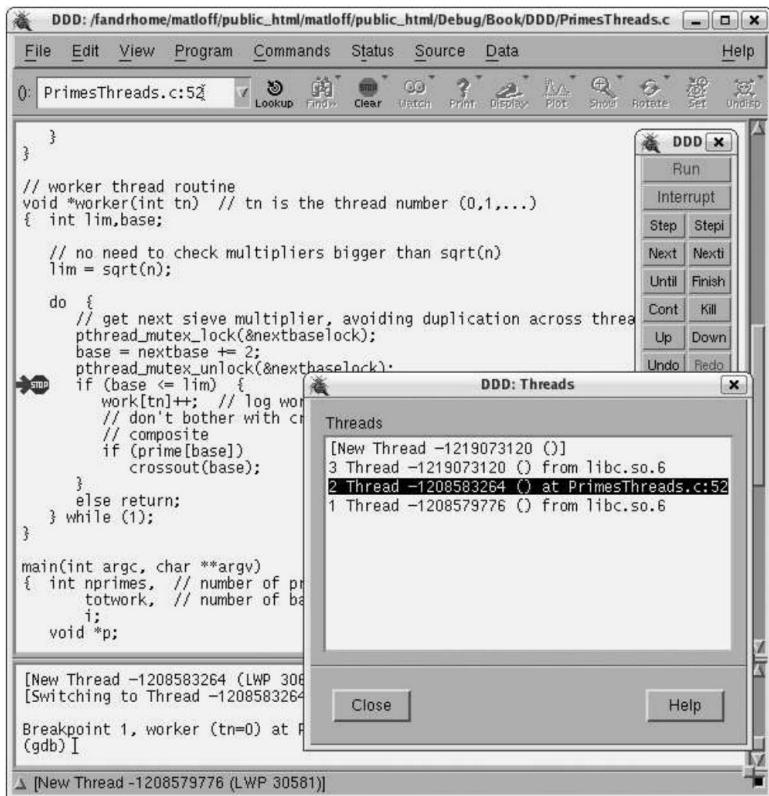


Рисунок 5-1: Окно тем

чтобы запросить операцию обратной трассировки, как это делается в DDD; стек вызовов отображается в списке потоков. Это изображено на рисунке 5-2. Как и выше, мы запустили программу на некоторое время, а затем прервали ее, нажав на значок «Приостановить», чтобы справа от Resume. Список потоков находится в представлении Debug, которое обычно в верхней левой части экрана, но здесь отображается в развернутом виде так как мы нажали «Развернуть» на вкладке «Отладка». (Мы можем нажать «Восстановить», чтобы вернуться к стандартной компоновке.)

Мы видим, что поток 3 был запущен во время прерывания; он получил сигнал SIGINT, который и является прерыванием (CTRL-C) сигнал. Мы также видим, что связанный системный вызов был вызван pthread_join(), который в свою очередь был вызван main(). Из того, что мы имеем видел об этой программе ранее, мы знаем, что это действительно главная нить.

Чтобы просмотреть информацию о другой теме, просто нажмите на треугольник рядом с нитью, чтобы направить ее вниз. Чтобы перейти на другую нить, мы щелкните по его записи в списке.

Мы можем захотеть установить точку останова, которая будет применяться только к определенному потоку. Для этого нам нужно сначала дождаться создания потока. Затем при выполнении

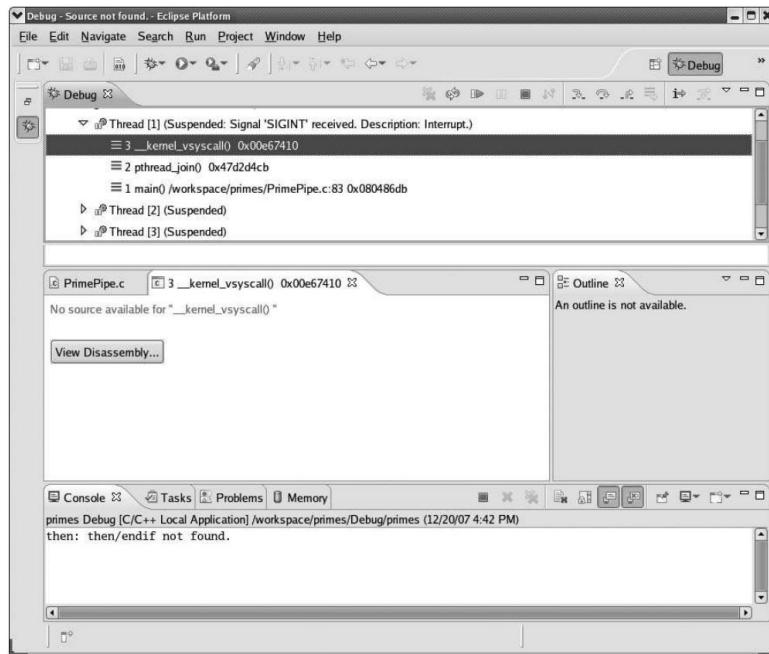


Рисунок5-2: Отображение тем в Затмение

приостанавливает работу посредством предыдущей установки точки останова или прерывания, как указано выше, мы щелкните правой кнопкой мыши по символу точки останова таким же образом, как если бы мы делали это, условная точка останова, но на этот раз выберите Фильтрация. Всплывающее окно типа появится тот, что на рисунке 5-3 . Мы видим, что эта точка останова в настоящее время применяется ко всем трем потокам. Если мы хотим, чтобы она применялась только к потоку 2, например, мы бы сняли флагки рядом с записями для двух других потоков.

5.3 Отладка параллельных приложений

Существует два основных типа архитектур параллельного программирования — общие память и передача сообщений.

Термин «разделяемая память» означает именно это: Несколько ЦП имеют доступ к некоторой общей физической памяти. Код, работающий на одном ЦП, взаимодействует с кодом, работающим на других, считывая и записывая данные в этот общая память, подобно тому, как потоки в многопоточном приложении взаимодействуют друг с другом через общее адресное пространство. (Действительно, потоковое программирование стало стандартным способом написания кода приложения для (Системы с общей памятью.)

Напротив, в среде передачи сообщений код, работающий на каждом ЦП может получить доступ только к локальной памяти этого ЦП, и он взаимодействует с другие отправляют строки байтов, называемые сообщениями, по каналу связи среда. Обычно это какая-то сеть, работающая либо на общем уровне,

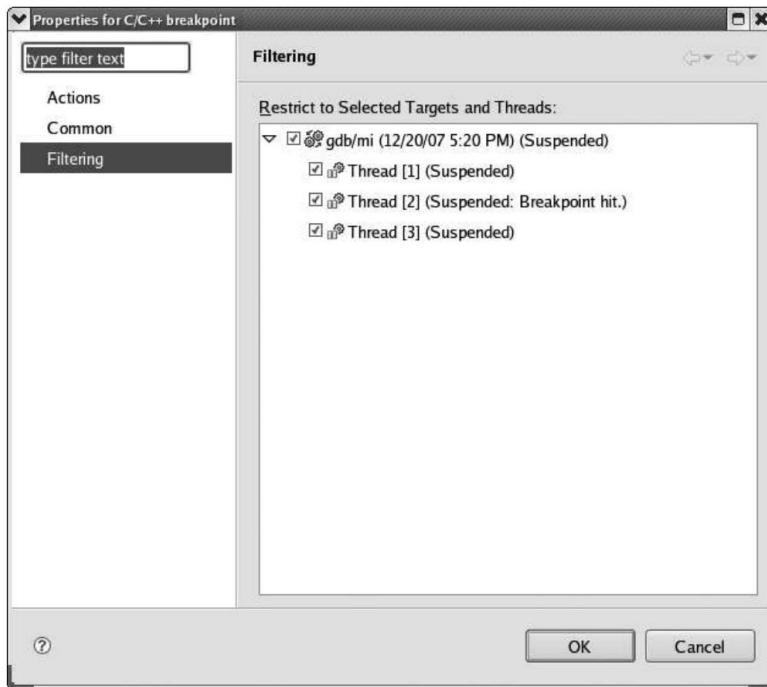


Рисунок 5-3: Параметр ^a точки останова, специфичная для потока Затмение

протокол назначения, такой как TCP/IP или специализированная программная инфраструктура, которая адаптирован для приложений передачи сообщений.

5.3.1 Передача сообщений Системы

Сначала мы обсудим передачу сообщений, используя в качестве примера популярный пакет Message Passing Interface (MPI). Мы используем реализацию MPICH

здесь, но те же принципы применимы к LAM и другим реализациям MPI.

Давайте снова рассмотрим программу нахождения простых чисел:

```

1 #включить <mpi.h>
2
3 // Пример программы MPI; не претендует на эффективность; находит и сообщает
4 // количество простых чисел, меньших или равных n
5
6 // Использует конвейерный подход: узел 0 просматривает все нечетные числа (т.е.
7 // мы предполагаем, что числа, кратные 2, уже отфильтрованы) и отфильтровываем
8 // те, которые кратны 3, передавая остальные в узел 1; узел 1
9 // отфильтровывает числа, кратные 5, передавая остальные в узел 2; узел 2
10 // отфильтровывает остальные составные элементы и затем сообщает число
11 // простых чисел
12

```

```

13 // аргументы командной строки — n и debugwait
14
15 #define PIPE_MSG 0 // тип сообщения, содержащего число для
16 // быть проверенным
17 #define END_MSG 1 // тип сообщения, указывающий, что больше данных не будет
18 // придет
19
20 int nnodes, // количество узлов в вычислении
21     n, // найти все простые числа от 2 до n
22     я; // мой номер узла
23
24 init(целое число argc,символ **argv)
25 { int debugwait; // если 1, то цикл продолжается до тех пор, пока
26 // отладчик был подключен
27
28     MPI_Init(&argc,&argv);
29     n = atoi(argv[1]);
30     debugwait = atoi(argv[2]);
31
32     MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
33     MPI_Comm_rank(MPI_COMM_WORLD,&me);
34
35     в то время как (debugwait) ;
36 }
37
38 недействительный узел0()
39 { int i,пустышка,
40     tocheck; // текущий номер для проверки перед передачей следующему узлу
41     для (i = 1; i <= n/2; i++) {
42         проверить = 2 * i + 1;
43         если (tocheck > n) перерыв;
44         если (проверить % 3 > 0)
45             MPI_Send(&tocheck,1,MPI_INT,1,PIPE_MSG,MPI_COMM_WORLD);
46     }
47     MPI_Send(&dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
48 }
49
50 недействительный узел1()
51 { int tocheck, // текущий номер для проверки из узла 0
52     дурачок;
53     MPI_Status статус; // см. ниже
54
55     в то время как (1) {
56         MPI_Recv(&tocheck,1,MPI_INT,0,MPI_ANY_TAG,
57                  MPI_COMM_WORLD,&статус);
58         если (status.MPI_TAG == END_MSG) перерыв;
59         если (проверить % 5 > 0)

```

```

60         MPI_Send(&tocheck,1,MPI_INT,2,PIPE_MSG,MPI_COMM_WORLD);
61     }
62     // теперь отправим наш сигнал об окончании данных, который передается в
63     // тип сообщения, а не само сообщение
64     MPI_Send(&dummy,1,MPI_INT,2,END_MSG,MPI_COMM_WORLD);
65   }
66
67 недействительный узел2()
68 { int tocheck, // текущий номер для проверки из узла 1
69     простое число,i,составной;
70     MPI_Status статус;
71
72     primecount = 3; // необходимо учитывать простые числа 2, 3 и 5, которые
73     // не будет обнаружен ниже
74     в то время как (1) {
75         MPI_Recv(&tocheck,1,MPI_INT,1,MPI_ANY_TAG,
76                 MPI_COMM_WORLD,&статус);
77         если (status.MPI_TAG == END_MSG) перерыв;
78         iscomposite = 0;
79         для (i = 7; i*i <= tocheck; i += 2)
80             если (проверить % i == 0) {
81                 iscomposite = 1;
82                 перерыв;
83             }
84             если (!iscomposite) простоечисло++;
85     }
86     printf("количество простых чисел = %d\n",primecount);
87 }
88
89 main(целое число argc,символ **argv)
90 { init(argc,argv);
91     переключить (меня) {
92         случай 0: узел0();
93         перерыв;
94         случай 1: узел1();
95         перерыв;
96         случай 2: node2();
97     };
98     MPI_Finalize();
99 }
```

Как пояснялось в комментариях в начале программы, здесь
наше Решето Эратосфена работает на трех узлах параллельной системы и
конвейерным способом. Первый узел начинается с нечетных чисел и удаляет
все кратные 3, передавая оставшиеся значения; второй узел принимает
выход первого и удаляет все кратные 5; и третий узел

берет выход второго, удаляет оставшиеся непростые числа и повторно сообщает количество оставшихся простых чисел.

Здесь конвейеризация достигается за счет того, что каждый узел передает одно число за раз к разу. (Гораздо большей эффективности можно было бы достичь, передавая групп чисел в каждом сообщении MPI, что сокращает объем коммуникаций накладные расходы.) При отправке числа на следующий узел, узел отправляет сообщение типа PIPE_MSG. Когда узла больше нет чисел для отправки, он сообщает об этом, отправляя сообщение типа END_MSG.

В качестве примера отладки предположим, что мы забыли включить последнее уведомление в первый узел, то есть мы забыли строку 46 в коде для узел0():

```
MPI_Send(&dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
```

Программа будет зависать на «нижестоящих» узлах. Посмотрим, как мы можем отследить эту ошибку. (Имейте в виду, что некоторые номера строк в GDB сеанс ниже будут отличаться на 1 от приведенного выше списка.)

Вы запускаете прикладную программу MPICH, вызывая скрипт с именем mpirun на одном узле системы. Затем скрипт запускает приложение программа на каждом узле через SSH. Здесь мы сделали это в сети из трех машин, которые мы назовем Узел 0, Узел 1 и Узел 2, с п, равным 100. Ошибка приводит к зависанию программы на последних двух узлах. Программа также зависает на первом узле, поскольку ни один экземпляр программы MPI не завершит работу пока все не выполнят функцию MPI_FINALIZE() .

Мы хотели бы использовать GDB, но поскольку мы использовали mpirun для вызова приложения на каждом из трех узлов, а не запускали их напрямую на узлы, мы не можем запустить GDB напрямую. Однако GDB позволяет динамически присоединить отладчик к уже запущенному процессу, используя процесс номер. Давайте запустим ps на узле 1, чтобы определить номер процесса который выполняет там наше приложение:

\$ ps aux		
...		
2755 ?	C 0:00 tcsh -c /home/matloff/primepipe	узел 1 3
2776 ?	C 0:00 /home/matloff/primepipe node1	32812 4
2777 ?	C 0:00 /home/matloff/primepipe node1	32812 4

Программа MPI выполняется как процесс 2776, поэтому мы подключаем GDB к программа на узле 1:

\$ gdb primepipe 2776	
...	
0xfffffe002 в ?? ()	

Это не очень информативно! Итак, посмотрим, где мы находимся:

(гdb) бт	
#0 0xfffffe002 в ?? ()	

```
#1 0x08074a76 в recv_message () #2
0x080748ad в p4_recv () #3 0x0807ab46
в MPID_CH_Check_incoming () #4 0x08076ae9 в
MPID_RecvComplete () #5 0x0806765f в
MPID_RecvDatatype () #6 0x0804a29f в PMPI_Recv
() #7 0x08049ce8 в node1 () в
PrimePipe.c:56 #8 0x08049e19 в main (argc=8,
argv=0xbffffb24) в PrimePipe.c:94 #9 0x420156a4 в __libc_start_main () из /lib/tls/libc.so.6
```

Из кадра 7 видно, что программа зависла на строке 56, ожидая получения от узла 0.

Далее было бы полезно узнать, какой объем работы был выполнен функцией, запущенной в узле 1, node1(). Она только началась или почти завершена?

Мы можем оценить прогресс, определив последнее обработанное значение для переменной tocheck:

```
(gdb) кадр 7 #7
0x08049ce8 в node1 () в PrimePipe.c:56
p MPI_Recv(&tocheck,1,MPI_INT,0,MPI_ANY_TAG, 56 (gdb)
tocheck $1 = 97
```

ПРИМЕЧАНИЕ Сначала нам нужно было перейти к кадру стека для node1(), используя команду GDB frame .

Это означает, что узел 1 находится в конце выполнения, так как 97 должно быть последним числом, которое узел 0 передает ему для проверки на простоту. Таким образом, в настоящее время мы ожидаем сообщение от узла 0 типа END_MSG. Тот факт, что программа зависла, предполагает, что узел 0 мог не отправить такое сообщение, что, в свою очередь, заставило бы нас проверить, отправил ли он его. Таким образом, мы надеемся быстро сосредоточиться на ошибке, которая заключалась в случайном пропуске строки 46.

Кстати, имейте в виду, что когда GDB вызывается командой

```
$ gdb primepipe 2776
```

как мы делали выше, обработка командной строки GDB сначала проверяет наличие файла ядра с именем 2776. В маловероятном случае, если такой файл существует, GDB загрузит его вместо присоединения к предполагаемому процессу. В качестве альтернативы, GDB также имеет команду присоединения .

В этом примере ошибка привела к зависанию программы. Подход к отладке параллельной программы, подобной этой, несколько отличается, когда симптомом является неправильный вывод. Предположим, например, что в строке 71 мы неправильно инициализировали primecount значением 2 вместо 3. Если мы попытаемся следовать той же процедуре отладки, программы, работающие на каждом узле, завершат выполнение и выйдут слишком быстро, чтобы вы успели подключить GDB. (Правда, мы могли бы использовать очень большое значение n, но обычно лучше сначала отлаживать простые случаи.) Нам нужно какое-то устройство, которое можно использовать, чтобы заставить программы ждать и

дать вам возможность присоединить GDB. Это цель строки 34 в функции init().

Как видно из исходного кода, значение debugwait берется из командной строки, предоставленной пользователем, где 1 означает ожидание, а 0 означает отсутствие ожидания. Если мы укажем 1 для значения debugwait, то когда каждый вызов программы достигнет строки 34, он останется там. Это даст нам время для присоединения GDB. Затем мы сможем выйти из бесконечного цикла и приступить к отладке.

Вот что мы делаем в узле 0:

```
узел1:~$ gdb primepipe 3124
...
0x08049c53 в init (argc=3, argv=0xbffffe2f4) в PrimePipe.c:34 while (debugwait); 34 (gdb) set
debugwait = 0 (gdb) c
```

Продолжение.

Обычно мы боимся бесконечных циклов, но здесь мы намеренно устанавливаем один, чтобы облегчить отладку. Мы делаем то же самое в узле 1 и узле 2, а в последнем мы также пользуемся возможностью установить точку останова на строке 77, прежде чем продолжить:

```
[matloff@node3 ~]$ gdb primepipe 2944
34          while (debugwait); (gdb)
b 77 Точка
останова 1 по адресу 0x8049d7d: файл PrimePipe.c, строка 77. (gdb)
set debugwait = 0 (gdb) c
```

Продолжение.

Точка останова 1, узел 2 () в PrimePipe.c:77

```
77          если (status.MPI_TAG == END_MSG) break; (gdb) p
для проверки $1 =
7 (gdb)
n 78
        iscomposite = 0;
(gdb) n 79
        для (i = 7; i*i <= tocheck; i += 2)
(gdb) n 84
        если (!iscomposite) простоечисло++;
(gdb) n 75
        MPI_Recv(&tocheck, 1, MPI_INT, 1, MPI_ANY_TAG, (gdb) p
простое_число $2 = 3
```

На этом этапе мы замечаем, что число простых чисел должно быть равно 4, а не 3 (простые числа от 7 до 7 — это 2, 3, 5 и 7), и таким образом мы нашли место ошибки.

5.3.2 Совместно используемая память Системы

А что насчет типа параллельного программирования с разделяемой памятью? Здесь у нас есть отдельные случаи для настоящих машин с разделяемой памятью и программно-распределенных настроек разделяемой памяти.

5.3.2.1 Настоящая общая память

Как упоминалось ранее, в среде с настоящей общей памятью прикладные программы часто разрабатываются с использованием потоков. Тогда применим наш материал в разделе 5.2 по отладке с помощью GDB/DDD.

OpenMP стал популярной средой программирования на таких машинах. OpenMP предоставляет программисту высокоуровневые конструкции параллельного программирования, которые в свою очередь используют потоки. Программист по-прежнему имеет доступ на уровне потоков, если это необходимо, но по большей части потоковая реализация директив OpenMP в значительной степени прозрачна для программиста.

В разделе 5.4 мы представляем расширенный пример отладки приложения OpenMP.

5.3.2.2 Программные распределенные системы с общей памятью

Цены на машины с двухъядерными процессорами теперь доступны обычным потребителям, но крупномасштабные системы с общей памятью со многими процессорами по-прежнему стоят сотни тысяч долларов. Популярной и недорогой альтернативой является сеть рабочих станций (NOW). Архитектуры NOW используют базовую библиотеку, которая создает иллюзию общей памяти. Библиотека, которая в значительной степени прозрачна для прикладного программиста, участвует в сетевых транзакциях, которые поддерживают согласованность копий общих переменных на разных узлах.

Этот подход называется программной распределенной общей памятью (SDSM). Наиболее широко используемая библиотека SDSM — Treadmarks, разработанная и поддерживаемая Университетом Райса. Другой превосходный пакет — JAIJA, доступный в Китайской академии наук (<http://www-users.cs.umn.edu/~tianhe/paper/dist.htm>).

Приложения SDSM демонстрируют определенные типы поведения, которые могут сбить с толку неосторожного программиста. Они сильно зависят от конкретной системы, поэтому здесь невозможно дать общее описание, но мы кратко обсудим несколько общих для многих из них вопросов.

Многие SDSM основаны на страницах, что означает, что они полагаются на базовое оборудование виртуальной памяти в узлах. Действия сложны, но мы можем дать краткий обзор. Рассмотрим переменную X, которая должна быть общей для узлов NOW. Программист указывает на это намерение, делая определенный вызов библиотеки SDSM, которая, в свою очередь, делает определенный системный вызов Unix, запрашивающий ОС заменить ее собственный обработчик ошибок сегмента на функцию в библиотеке SDSM для ошибок страниц, включающих страницу, содержащую X. SDSM настраивает все таким образом, что только узлы NOW с действительными копиями X имеют соответствующие страницы памяти, помеченные как резидентные. Когда X обращается к какому-либо другому узлу, возникает ошибка страницы, и базовое программное обеспечение SDSM извлекает правильное значение из узла, который его имеет.

Опять же, не обязательно знать точные принципы работы системы SDSM; важно просто понимать, что существует некая неопределенность.

derlying VM-based mechanism, который используется для поддержания согласованности локальных копий общих данных на узлах NOW. Если вы этого не сделаете, вы будете озадачены, когда попытаетесь отладить код приложения SDSM. Отладчик будет казаться таинственным образом останавливающимся для несуществующих ошибок сегментации, потому что инфраструктура SDSM намеренно генерирует ошибки сегментации, и когда программа приложения SDSM запускается под инструментом отладки, инструмент их обнаруживает. Как только вы это поймете, проблем вообще не будет — в GDB вы просто выдадите команду `continue`, чтобы возобновить выполнение, когда произойдет одна из этих странных пауз.

У вас может возникнуть соблазн приказать GDB не останавливаться или выдавать предупреждающие сообщения. Sages при возникновении любых ошибок сегмента, используя команду GDB

обрабатывать SIGSEGV nostop noprint

Однако этот подход следует использовать с осторожностью, поскольку он может привести к пропуску реальных ошибок сегментации, вызванных ошибками в прикладной программе.

Еще одна связанная с этим трудность отладки приложений, работающих на страничных SDSM, возникает следующим образом. Если узел в сети изменяет значение общей переменной, то любой другой узел, которому нужно значение этой переменной, должен получить обновленное значение через сетевую транзакцию. Опять же, детали того, как это происходит, зависят от системы SDSM, но это означает, что если вы пошагово проходите код, выполняемый на одном узле, вы можете обнаружить, что GDB загадочным образом зависит, потому что узел теперь ждет обновления своей локальной копии переменной, которая была недавно изменена другим узлом. Если вы случайно запустили отдельный сеанс GDB для пошагового прохождения кода на этом другом узле, обновление не произойдет на первом узле, пока сеанс отладки на втором узле не продвинется достаточно далеко. Другими словами, если программист не будет бдителен и осторожен во время отладки приложения SDSM, он может вызвать собственную ситуацию тупика через сам процесс отладки.

Ситуация с SDSM похожа на случай передачи сообщений в одном смысле — необходимость иметь переменную, такую как `debugwait` в примере MPI выше, которая позволяет приостанавливать программу на всех узлах, давая вам возможность присоединить GDB к каждому узлу и пройти программу с самого начала.

5.4 Расширенный пример

В этом разделе представлен пример отладки приложения с общей памятью, разработанного с использованием OpenMP. Необходимые знания OpenMP будут объяснены ниже. Все, что нужно, — это базовое понимание потоков.

5.4.1 OpenMP Обзор

OpenMP по сути является высокоуровневым интерфейсом параллельного программирования для операций управления потоками. Количество потоков задается через переменную окружения OMP_NUM_THREADS. В оболочке C, например, вы вводите

```
% setenv OMP_NUM_THREADS 4
```

в командной строке настроить четыре потока.

Код приложения состоит из кода на языке C, перемежаемого директивами OpenMP.

Каждая директива применяется к блоку, который следует за ней, разделенному левой и правой фигурными скобками. Самая простая директива —

```
#pragma omp параллель
```

Это устанавливает OMP_NUM_THREADS потоков, каждый из которых одновременно выполняет блок кода, следующий за прагмой. Обычно в этот блок встроены другие директивы.

Другая очень распространенная директива OpenMP —

```
#pragma omp барьер
```

Это определяет «точку встречи» для всех потоков. Когда какой-либо поток достигает этой точки, он блокируется до тех пор, пока все остальные потоки не прибудут туда.

Часто может возникнуть необходимость, чтобы определенный блок выполнялся только одним потоком, в то время как другие потоки пропускают его. Это достигается путем записи

```
#pragma omp сингл
```

Сразу за таким блоком следует подразумеваемый барьер.

Существует множество других директив OpenMP, но в этом примере используется только одна:

```
#pragma omp критический
```

Как следует из названия, это создает критическую секцию, в которой только один обсуждение разрешено в любое время.

5.4.2 OpenMP Пример Программа

Мы реализуем знаменитый алгоритм Дейкстры для определения минимальных расстояний между парами вершин во взвешенном графе. Допустим, нам даны расстояния между соседними вершинами (если две вершины не являются соседними, расстояние между ними устанавливается равным бесконечности). Цель состоит в том, чтобы найти минимальные расстояния между вершиной 0 и всеми остальными вершинами.

Ниже приведен исходный файл `dijkstra.c`. Он генерирует случайные длины ребер среди указанного количества вершин, а затем находит минимальные расстояния от вершины 0 до каждой из других вершин.

1 // дейкстра.c

2

3 // Пример программы OpenMP: поиск кратчайшего пути Дейкстры в двунаправленном
графе 4 ; находит кратчайший путь из вершины 0 во все

```

5 // другие
6
7 // использование: dijkstra nv print
8

9 // где nv — размер графика, а print — 1, если график и min
10 // расстояния должны быть выведены, в противном случае 0
11
12 #include <omp.h> // обязательно
13 #include <значения.h>
14
15 // включение stdlib.h и stdio.h, похоже, вызывает конфликт с
16 // Omni-компилятором, поэтому объявляйте напрямую
17 внешний void *malloc();
18 extern int printf(char *...);
19
20 // глобальные переменные, общие для всех потоков
21 int nv, // количество вершин
22     *notdone, // вершины еще не проверены
23     nth, // количество потоков
24     chunk, // количество вершин, обработанных каждым потоком
25     md, // текущий минимум по всем потокам
26     mv; // вершина, которая достигает этого минимума
27
28 int *ohd, // 1-скаковые расстояния между вершинами; "ohd[i][j]" - это
29     // охд[i*nv+j]
30     *mind; // минимальные расстояния, найденные на данный момент
31
32 void init(int ac, char **av)
33 { int i,j,tmp;
34     nv = atoi(av[1]);
35     ohd = malloc(nv*nv*sizeof(int));
36     разум = malloc(nv*sizeof(int));
37     notdone = malloc(nv*sizeof(int));
38     // случайный график
39     для (i = 0; i < nv; i++)
40         для (j = i; j < nv; j++) {
41             если (j == i) ohd[i*nv+i] = 0;
42             еще {
43                 ohd[nv*i+j] = rand() % 20;
44                 охд[nv*j+i] = охд[nv*i+j];
45             }
46         }
47     для (i = 1; i < nv; i++) {
48         невыполнено[i] = 1;
49         разум[i] = охд[i];
50     }
51 }

```

```

52
53 // находит ближайшее к 0 среди notdone, среди s через e; возвращает min
54 // расстояние в *d, ближайшая вершина в *v
55 void findmymin(целое s, целое e, целое *d, целое *v)
56 { int i;
57     *d = МАКСИНТ;
58     для (i = s; i <= e; i++)
59         если (не сделано[i] && разум[i] < *d) {
60             *d = разум[i];
61             *v = я;
62         }
63     }
64
65 // для каждого i в {s,...,e}, спросить, существует ли более короткий путь к i, через
66 // mb
67 void updatemind(целое s, целое e)
68 { int i;
69     для (i = s; i <= e; i++)
70         если (не сделано[i])
71             если (разум[mb] + охд[mb*nv+i] < разум[i])
72                 разум[i] = разум[mv] + охд[mv*nv+i];
73     }
74
75 недействительный dowork()
76 {
77     #pragma omp параллель
78     { int startv,endv, // начальная и конечная вершины для этого потока
79         шаг, // вся процедура проходит nv шагов
80         тумв, // вершина, которая достигает этого значения
81         я = omp_get_thread_num(),
82         тумд; // минимальное значение, найденное этим потоком
83     #pragma omp сингл
84     { nth = omp_get_num_threads(); кусок = nv/nth;
85         printf("существует %d потоков\n",nth); }
86     startv = я * кусок;
87     конец_v = начало_v + кусок - 1;
88     // алгоритм проходит nv итераций
89     для (шаг = 0; шаг < nv; шаг++) {
90         // найти ближайшую вершину к 0 среди notdone; каждый поток находит
91         // ближайший в своей группе, то находим общий ближайший
92         #pragma omp сингл
93         { md = МАКСИНТ;
94             mb = 0;
95         }
96         findmymin(startv,endv,&тумд,&тумв);
97         // обновить общий минимум, если мой меньше
98         #pragma omp критический

```

```

99         { если (моймд < мд)
100             { md = моймд; }
101         }
102         #pragma omp барьер
103         // отметить новую вершину как выполненную
104         #pragma omp сингл
105         { невыполнено[mv] = 0; }
106         // теперь обновлю свой раздел оhd
107         обновление_разума(startv,endv);
108     }
109 }
110 }
111
112 int main(int argc, char **argv)
113 {
114     int i,j,print;
115     init(argc,argv);
116     // начать параллельно
117     делать работу();
118     // вернуться к одиночному потоку
119     печать = atoi(argv[2]);
120     если (печатать) {
121         printf("веса графика:\n");
122         для (i = 0; i < nv; i++) {
123             для (j = 0; j < nv; j++)
124                 printf("%u ",ohd[nv*i+j]);
125             printf("\n");
126         }
127         printf("минимальные расстояния:\n");
128         для (i = 1; i < nv; i++)
129             printf("%u\n",mind[i]);
130     }
}

```

Давайте рассмотрим, как работает алгоритм. Начнем со всех вершин, кроме вершины 0, которая в данном случае является вершинами с 1 по 5, в наборе «не сделано». В каждом итерации алгоритма, выполните следующее:

1. Найдите «незавершенную» вершину v , которая находится ближе всего к вершине 0, по известным путям. пока что. Эта проверка выполняется всеми потоками, каждый поток проверяет одинаковое количество вершин. Функция, которая выполняет эту работу, — `findmymin()`.
2. Затем переместите v в набор «выполнено».
3. Для всех оставшихся вершин i в наборе «не сделано» проверьте, происходит ли сначала от 0 до v по наиболее известному на данный момент пути, а затем от v до i за один прыжок, короче текущего кратчайшего расстояния от 0 до i . Если так что обновите это расстояние соответственно. Функция, которая выполняет эти действия — это `updatemind()`.

Итерация продолжается до тех пор, пока набор «не выполнено» не станет пустым.

Поскольку директивы OpenMP требуют предварительной обработки, всегда существует потенциальная проблема, что мы потеряем наши исходные номера строк и имена переменных и функций. Чтобы увидеть, как это решить, мы обсудим два разных компилятора. Сначала мы рассмотрим компилятор Omni (<http://www.hpcc.jp/Omni/>), а затем GCC (требуется версия 4.2 или более поздняя).

Мы компилируем наш код под Omni следующим образом:

```
$ omcc -g -o dij dijkstra.c
```

После компиляции программы и запуска ее с четырьмя потоками мы обнаруживаем, что она работает неправильно:

```
$ dij 6 1
есть 4 потока график
веса: 03617 15 13
3015612 9 6 150127
1761010 19
```

```
15 1221003
13971930
минимальные расстояния:
3
6
17
15
13
```

Анализ графика вручную показывает, что правильные минимальные расстояния должно быть 3, 6, 7, 8 и 11.

Далее запускаем программу в GDB. Здесь очень важно понимать, стоять на последствиях того факта, что OpenMP работает через директивы. Хотя номера строк, имена функций и т. д. в основном сохраняются двумя обсуждаемыми здесь компиляторами, между ними есть некоторые расхождения.

Посмотрите, что происходит, когда мы пытаемся установить точку останова в исполняемом файле dij в начале сеанса GDB:

```
(gdb) tb основной
Точка останова 1 по адресу
0x80492af (gdb) r 6 1
Запуск программы: /debug/dij 6 1
[Отладка потоков с использованием libthread_db включена]
[Новая тема -1208490304 (LWP 11580)]
[Переключение на поток -1208490304 (LWP 11580)]
0x080492af в main () (gdb)
|
```

```
1      /tmp/omni_C_11486.c: Такого файла или каталога нет. в /
      tmp/omni_C_11486.c
```

Мы обнаруживаем, что точка останова находится не в исходном файле. Вместо этого она находится в коде инфраструктуры OpenMP Omni. Другими словами, `main()` здесь — это `main()` Omni, а не ваш собственный. Компилятор Omni искажил имя нашего `main()` до `_ompc_main()`.

Чтобы установить точку останова в `main()`, мы вводим

```
(gdb) tb _ompc_main
Точка останова 2 по адресу 0x80491b3: файл dijkstra.c, строка 114.
```

и проверьте это, продолжив:

```
(gdb) c
Продолжение.
[Новая тема -1208493152 (LWP 11614)]
[Новая тема -1218983008 (LWP 11615)]
[Новая тема -1229472864 (LWP 11616)]
_ompc_main (argc=3, argv=0xbfbab6314) в dijkstra.c:114 init(argc,argv);
114
```

Хорошо, вот знакомая строка `init()`. Конечно, мы могли бы дать команду

```
(gdb) б dijkstra.c:114
```

Обратите внимание на создание трех новых тем, в результате чего их стало четыре.

Однако, как бы мы ни решили установить точки останова, здесь нам придется проделать немного больше работы, чем обычно, поэтому особенно важно оставаться в пределах одного сеанса GDB между запусками программы, даже когда мы изменяем исходный код и перекомпилируем его, чтобы сохранить точки останова, условия и т. д.

Таким образом, нам придется потратить время на настройку всего этого один раз.

Теперь, как вы отслеживаете ошибку(и)? Естественно подойти к отладке этой программы, проверяя результаты в конце каждой итерации. Основные результаты находятся в наборе «не сделано» (в массиве `notdone[]`) и в текущем списке наиболее известных расстояний от 0 до других вершин, то есть массиве `mind[]`. Например, после первой итерации набор «не сделано» должен состоять из вершин 2, 3, 4 и 5, причем вершина 1 была выбрана в этом

итерация.

Вооружившись этой информацией, давайте применим принцип подтверждения и проверим `notdone[]` и `mind[]` после каждой итерации цикла `for` в `dowork()`.

Мы должны быть осторожны, когда устанавливаем точки останова. Хотя естественным местом для этого кажется строка 108, в самом конце основного цикла алгоритма, это может быть не так уж хорошо, поскольку GDB остановится там для каждого потока. Вместо этого выберите размещение точки останова внутри одного блока OpenMP, чтобы она остановилась только для одного потока.

Поэтому вместо этого мы проверяем результаты после каждой итерации, останавливаясь в начале цикла, начиная со второй итерации:

```
(gdb) b 92 если шаг >= 1
Точка останова 3 по адресу 0x80490e3: файл dijkstra.c, строка 92.
(gdb) c
Продолжение.
Имеется 4 потока
```

```
Точка останова 3, __omp_func_0 () в dijkstra.c:93 { md =
93           MAXINT;
```

Давайте убедимся, что первая итерация действительно выбрала правильную вершину (вершина 1) для перемещения из набора «не выполнено»:

```
(gdb) p mv
$1 = 0
```

Гипотеза не подтверждается, в конце концов. Проверка кода показывает, что в строке 100 мы забыли установить mv. Исправляем это так, чтобы читать

```
{ md = мойmd; mv = мойmv; }
```

Итак, мы перекомпилируем и снова запустим программу. Как было отмечено ранее в этом разделе (и в других местах этой книги), очень полезно не выходить из GDB при повторном запуске программы. Мы могли бы запустить программу в другом окне терминала, но просто для разнообразия давайте воспользуемся другим подходом. Мы временно отключаем наши точки останова, выполнив команду dis , затем запускаем перекомпилированную программу из GDB, а затем снова включаем точки останова, используя ена:

```
(gdb) dis
(gdb) r
Отлаживаемая программа уже запущена.
Начать с начала? (y или n) y `/debug/dij'
изменился; перечитываю символы.
Запуск программы: `/debug/dij' 6 1
[Отладка потоков с использованием libthread_db включена]
[Новая тема -1209026880 (LWP 11712)]
[Новая тема -1209029728 (LWP 11740)]
[Новая тема -1219519584 (LWP 11741)]
[Новая тема -1230009440 (LWP 11742)] есть 4
темы график веса: 03617
15 13

3015612 9
6 150127
1761010 19
```

```
15 1221003
13971930
минимальные расстояния:
3
6
17
15
13
```

Программа завершилась с кодом
06. (gdb) ena

Мы все еще получаем неправильные ответы. Давайте проверим все в этой точке останова еще раз:

```
(гдб) р
Запуск программы: /debug/dij 6 1
[Отладка потоков с использованием libthread_db включена]
[Новая тема -1209014592 (LWP 11744)]
[Новая тема -1209017440 (LWP 11772)]
[Новая тема -1219507296 (LWP 11773)]
[Новая тема -1229997152 (LWP 11774)] есть 4
темы
[Переключение на тему -1209014592 (LWP 11744)]
```

Точка останова 3, __omp_func_0 () в dijkstra.c:93 { md = MAXINT; 93

```
(гдб) п мв $2
= 1
```

По крайней мере mv теперь имеет правильное значение. Давайте проверим mind[]:

```
(gdb) p *mind@6
$3 = {0, 3, 6, 17, 15, 13}
```

Обратите внимание, что поскольку мы создали массив mind[], динамически с помощью malloc(), мы не могли использовать команду печати GDB в ее обычной форме. Вместо этого мы использовали функцию искусственного массива GDB.

В любом случае, mind[] все еще неверен. Например, mind[3] должно быть 3 + 6 = 9, но это 17. Давайте проверим код, который обновляет mind[]:

```
(gdb) b 107 if me == 1 Точка
останова 4 по адресу 0x8049176: файл dijkstra.c, строка 107. (гдб) r
```

Отлаживаемая программа уже запущена.
Начать с начала? (у или п) у Запуск программы: /
debug/dij 6 1 [Отладка потоков с
использованием libthread_db включена]

```
[Новая тема -1209039168 (LWP 11779)]
[Новая тема -1209042016 (LWP 11807)]
[Новая тема -1219531872 (LWP 11808)]
[Новая тема -1230021728 (LWP 11809)] есть
4 темы
[Переключение на тему -1230021728 (LWP 11809)]
```

```
Точка останова 4, __omp_func_0 () в dijkstra.c:107
107          updatemind(startv,endv);
```

Сначала убедитесь, что startv и endv имеют разумные значения:

```
(gdb) p начало
$4 = 1
(gdb) p конец
$5 = 1
```

Размер куска всего 1? Давайте посмотрим:

```
(gdb) p кусок $6
= 1
```

После проверки вычисления для chunk мы понимаем, что нам нужно количество потоков, чтобы равномерно разделить nv. Последнее имеет значение 6, которое не делится на наше количество потоков, 4. Мы делаем себе заметку, чтобы позже вставить какой-нибудь код для отлова ошибок, и уменьшаем наше количество потоков до 3 на данный момент.

И снова, нам не нужно выходить из GDB, чтобы сделать это. GDB наследует переменные окружения при первом вызове, но значения этих переменных также могут быть изменены или установлены в GDB, и это то, что мы здесь делаем:

```
(gdb) установить окружение OMP_NUM_THREADS = 3
```

Теперь давайте снова запустим:

```
(gdb) dis
(gdb) r
Отлаживаемая программа уже запущена.
Начать с начала? (y или n) у Запуск программы: /
debug/dij 6 1 [Отладка потоков с
использованием libthread_db включена]
[Новая тема -1208707392 (LWP 11819)]
[Новая тема -1208710240 (LWP 11847)]
[Новая тема -1219200096 (LWP 11848)] есть
3 темы
веса графика:
03617 15 13
3015612 9
6 150127
```

```
1761010 19  
15 1221003  
13971930  
минимальные расстояния:
```

```
3  
6  
7  
15  
12
```

Программа завершилась с кодом

```
06. (gdb) ena
```

Айя, все те же неправильные ответы! Продолжаем проверять процесс обновления для mind[]:

```
(гдб) р  
Запуск программы: /debug/dij 6 1  
[Отладка потоков с использованием libthread_db включена]  
[Новая тема -1208113472 (LWP 11851)]  
[Новая тема -1208116320 (LWP 11879)]  
[Новая тема -1218606176 (LWP 11880)] есть  
3 темы  
[Переключение на тему -1218606176 (LWP 11880)]
```

```
Точка останова 4, __ompc_func_0 () в dijkstra.c:107 updatemind(startv,endv);
```

```
107
```

```
(gdb) р начало  
$7 = 2  
(gdb) р конец  
$8 = 3
```

Хорошо, это правильные значения для startv и endv в случае me = 1. Итак, вводим функцию:

```
(гдб) с  
[Переключение на тему -1208113472 (LWP 11851)]
```

```
Точка останова 3, __ompc_func_0 () в dijkstra.c:93 { md = MAXINT; 93
```

```
(gdb) с  
Продолжение.  
[Переключение на тему -1218606176 (LWP 11880)]  
updatemind (s=2, e=3) в dijkstra.c:69 для (i = s; i <=  
e; i++) 69
```

Обратите внимание, что из-за переключений контекста между потоками мы не вошли updatemind() немедленно. Теперь проверим случай i=3:

```
(gdb) tb 71 if i == 3 Точка
останова 5 в 0x8048fb2: файл dijkstra.c, строка 71. (gdb) с
```

Продолжение.

```
updatemind (s=2, e=3) в dijkstra.c:71
71      если (разум[мв] + охд[мв*нв+и] < разум[i])
```

Как обычно, мы применяем Принцип Подтверждения:

```
(gdb) p mv
$9 = 0
```

Ну, это большая проблема. Вспомним, что в первой итерации тм оказывается быть 1. Почему здесь 0? Через некоторое время мы понимаем, что эти переключения контекста должны были быть большой намек. Взгляните на вывод GDB выше еще раз. Поток, системный идентификатор которого 11851, уже был на строке 93 — другими словами, он уже был на следующей итерации основного цикла алгоритма. Фактически, когда мы нажали с для продолжения, он даже выполнил строку 94, которая

```
мв = 0;
```

Этот поток перезаписал предыдущее значение тм, равное 1, так что поток, который обновляет mind[3], теперь полагается на неправильное значение тм. Решение состоит в том, чтобы добавить еще один барьер:

```
updatemind(startv,endv);
#pragma omp барьер
```

После этого исправления программа работает корректно.

Вышеизложенное было основано на компиляторе Omni. Как уже упоминалось, начиная с версии 4.2, GCC также обрабатывает код OpenMP. Все, что вам нужно сделать, это добавить флаг -fopenmp в командную строку GCC.

В отличие от Omni, GCC генерирует код таким образом, что фокус GDB находится на свой собственный исходный файл с самого начала. Таким образом, выдача команды

```
(gdb) б основной
```

в самом начале сеанса GDB действительно приведет к установке точки останова в собственном main(), в отличие от того, что мы видели для компилятора Omni.

Однако на момент написания статьи основным недостатком GCC является то, что символы для локальных переменных, которые находятся внутри параллельного блока OpenMP (называемых частными переменными в терминологии OpenMP), не будут видны в GDB. Например, команда

```
(гдб) п мв
```

которую вы ввели для кода, сгенерированного Omni выше, будет работать и для кода, сгенерированного GCC, но команда

(gdb) р startv

приведет к сбою кода, сгенерированного GCC.

Конечно, есть способы обойти это. Например, если вы хотите узнать значение startv, вы можете запросить значение s в updatemind().

Надеюсь, эта проблема будет решена в следующей версии GCC.

6

СПЕЦИАЛЬНЫЕ ТЕМЫ



Во время отладки возникают различные проблемы, которые не занимайтесь отладочными инструментами. Мы рассмотрим некоторые из этих вопросов в этой главе.

6.1 Что делать, если приложение даже не компилируется или не загружается?

GDB, DDD и Eclipse бесценны, но они не смогут вам помочь, если ваша программа даже не компилируется. В этом разделе мы дадим вам несколько советов, как справиться с этой ситуацией.

6.1.1 ФАНТОМ Линия Числа в Синтаксис Ошибки Сообщения

Иногда компилятор сообщает вам, что в строке x есть синтаксическая ошибка , когда на самом деле строка x совершенно верна, а настоящая ошибка находится в более ранней строке. Например, вот исходный файл bintree.c из Главы 3 с ошибкой синтаксиса, допущенной в точке, которую мы пока не будем раскрывать (ну, она довольно очевидно, если вы хотите это найти).

1 // bintree.c: процедуры для вставки и сортированной печати двоичного дерева

2

3 #include <stdio.h>

4 #включить <stdlib.h>

```

5
6 структурный узел {
7     int val; // сохраненное значение
8     struct node *left; // указатель на меньший дочерний элемент
9     struct node *right; // указатель на больший дочерний элемент
10    };
11
12 typedef структурный узел *nsp;
13
14 nsp корня;
15
16 nsp makenode(int x)
17  {
18      НСП ТМП;
19
20      tmp = (nsp) malloc(sizeof(struct node));
21      tmp->val = x;
22      tmp->влево = tmp->вправо = 0;
23      возврат tmp;
24  }
25
26 пустая вставка (nsp *btp, int x)
27  {
28      НСП ТМП = *бтп;
29
30      если (*бтп == 0) {
31          *бтп = makenode(x);
32          возвращаться;
33      }
34
35      в то время как (1)
36      {
37          если (x < tmp->val) {
38
39              если (tmp->left != 0) {
40                  tmp = tmp->left;
41              } еще {
42                  tmp->left = makenode(x);
43                  перерыв;
44              }
45          } еще {
46
47              если (tmp->right != 0) {
48                  tmp = tmp->right;
49              } еще {
50                  tmp->right = makenode(x);
51

```

```

52             перерыв;
53         }
54
55     }
56 }
57
58 пустое дерево печати (nsp bt)
59 {
60     если (bt == 0) вернуть;
61     printtree(bt->left);
62     printf("%d\n",bt->val);
63     printtree(bt->right);
64 }
65
66 int main(int argc, char *argv[])
67 {
68     целочисленный x;
69
70     корень = 0;
71     для (i = 1; i < argc; i++)
72         вставить(&корень, atoi(argv[i]));
73     printtree(корень);
74 }
```

Выполнение этого через GCC дает

```
$ gcc -g bintree.c
bintree.c: В функции `insert':
bintree.c:75: ошибка синтаксического анализа в конце ввода
```

Поскольку строка 74 является концом исходного файла, второе сообщение об ошибке: довольно неинформативно, мягко говоря. Но первое сообщение предполагает, что проблема в `insert()`, так что это подсказка, хотя там и не сказано, что именно в чем проблема.

В такой ситуации типичным виновником является отсутствующая закрывающая скобка или точка с запятой. Вы можете проверить это напрямую, но в большом исходном файле это может быть сложно. Давайте пойдем другим путем.

Вспомним принцип подтверждения из главы 1. Давайте сначала подтвердите, что проблема действительно в `insert()`. Для этого временно закомментируйте эту функцию из исходного кода:

```
...
tmp->val = x;
tmp->влево = tmp->вправо = 0;
возврат tmp;
}

// void insert(nsp *btp, int x)
```

```

// {
    НСП ТМП = *бтп;
// //
    если (*бтп == 0) {
// //
        *бтп = makenode(x);
// //
        возвращаться;
    }
//
//     в то время как (1)
{
//         если (x < tmp->val) {
//
//             если (tmp->left != 0) {
//                 tmp = tmp->left;
//             } еще {
//                 tmp->left = makenode(x);
//                 перерыв;
//             }
//
//             } еще {
//
//                 если (tmp->right != 0) {
//                     tmp = tmp->right;
//                 } еще {
//                     tmp->right = makenode(x);
//                     перерыв;
//                 }
//             }
//         }
//     }
}

void printtree(nsp bt)
{
    если (bt == 0) вернуть;
...

```

ПРИМЕЧАНИЕ. Предпочтительно использовать быструю комбинацию клавиш для комментирования, например, блок операция. Будут обсуждаться сочетания клавиш текстового редактора, полезные при отладке контекстов в Главе 7.

Сохраните файл, а затем повторно запустите GCC:

```
$ gcc -g bintree.c
/tmp/ccg0LDCS.o: В функции `main':
/home/matloff/public_html/matloff/public_html/Debug/Book/DDD/bintree.c:72:
неопределенная ссылка на `insert'
collect2: Id вернул 1 статус выхода
```

Не отвлекайтесь на тот факт, что компоновщик LD пожаловался, что не может найти `insert()`. В конце концов, вы знали, что это произойдет, так как вы закомментировали эту функцию. Вместо этого, интерес представляет то, что нет жалобы на синтаксическую ошибку, как это было раньше. Итак, вы действительно подтвердили, что синтаксическая ошибка находится где-то в `insert()`. Теперь раскомментируйте строки этой функции (опять же, желательно с помощью сочетания клавиш текстового редактора, например «отменить») и сохраните файл. Также, просто чтобы убедиться, что вы восстановили все правильно, повторно запустите GCC, чтобы убедиться, что синтаксическая ошибка снова появляется (здесь не показано).

На этом этапе вы можете применить другой принцип, изложенный в Главе 1: Принцип бинарного поиска. Повторно сужайте область поиска в функции `insert()`, каждый раз уменьшая ее вдвое, пока не получите достаточно маленькую область, в которой можно обнаружить синтаксическую ошибку.

Для этого сначала закомментируйте примерно половину функции. Разумным способом сделать это было бы просто закомментировать цикл `while`. Затем перезапустите GCC:

```
$ gcc -g bintree.c $
```

Aha! Сообщение об ошибке исчезло, так что проблема с синтаксисом должна быть где-то внутри цикла. Итак, вы сузили проблему до этой половины функции, и теперь вы сократите эту область еще на половину. Для этого закомментируйте код `else`:

```
недействительная вставка(nsp *btp, int x)
{
    НСП tmp = *бтп;

    если (*бтп == 0)
        { *бтп = makenode(x);
        return;
    }

    в то время как
    (1) {
        если (x < tmp->val) {

            если (tmp->left != 0) { tmp
                = tmp->left; } иначе
            { tmp-
                >left = makenode(x); break;

            }

        } // еще {
    //
    //      если (tmp->right != 0) {
```

```

        tmp = tmp->right;
    } еще {
        tmp->right = makenode(x);
        перерыв;
    //////////////////////////////////////////////////////////////////}
    //
    //}
}

```

Повторно запустив GCC, вы обнаружите, что проблема появляется снова:

```
$ gcc -g bintree.c
bintree.c: В функции `insert':
bintree.c:75: ошибка синтаксического анализа в конце ввода
```

Итак, синтаксическая ошибка либо в блоке `if`, либо в конце функции. К этому времени вы сузили проблему до семи строк код, поэтому вы, вероятно, сможете найти проблему путем визуального осмотра; оказывается, мы случайно пропустили закрывающую скобку в внешний если-то-иначе.

Принцип бинарного поиска может быть очень полезен при поиске синтаксических ошибок. Их неизвестных мест. Но временно комментируя код, будьте уверены, что вы не создаете новые синтаксические ошибки сами! Закомментируйте целый функция, целый цикл и т. д., как мы сделали здесь.

6.1.2 Отсутствующий Библиотеки

Иногда GCC — на самом деле LD, компоновщик, который вызывается GCC во время процесса построения вашей программы — сообщит вам, что он не может найти одна или несколько функций,ываемых вашим кодом. Обычно это происходит из-за отказа информировать GCC о местоположении библиотек функций. Многие, если не большинство, читатели этой книги будут хорошо разбираться в этой теме, но для тех, кто не разбирается, мы дадим краткое введение в этом разделе. Обратите внимание, что наше обсуждение здесь применимо в основном к Linux и в различной степени к другим системам семейства Unix операционные системы.

6.1.2.1 Пример

Давайте используем в качестве примера следующий очень простой код, состоящий из основного programma, в ac,

```
// ac
```

```
int f(int x);
```

```
основной()
{
    int v;
```

```

scanf("%d",&v);
printf("%d\n",f(v));
}

```

и подпрограмма в z/bc:

```
// до нашей эры
```

```
целочисленный ю целочисленный
```

```
x {
    вернуть x*x;
}
```

Если вы попытаетесь скомпилировать ac без попытки скомпоновать код в bc, то ЛД, конечно, будет жаловаться:

```
$ gcc -g ac /tmp/
ccIP5WHu.o: В функции `main': /debug/ac:9:
неопределенная ссылка на `f' collect2: ld вернул 1
статус выхода
```

Мы могли бы перейти к z, скомпилировать bc , а затем подключить объектный файл:

```
$ cd z
$ gcc -g -c bc $ cd
$ gcc ..
-g ac z/bc
```

Однако, если вам нужно связать много функций, возможно, из разных исходные файлы, и если эти функции, вероятно, будут полезны для будущих программ, которые вы можете написать, вы можете создать библиотеку, один архивный файл. Существует два типа библиотечных файлов. Когда вы компилируете код, который вызывает функции в статической библиотеке, эти функции становятся частью результирующего исполняемого файла. С другой стороны, если библиотека является динамической, функции физически не привязаны к вызывающему коду до тех пор, пока программа не будет фактически выполнена.

Вот как можно создать статическую библиотеку, скажем, lib88.a, для приведенного здесь примера.

ПРИМЕЧАНИЕ В системах Unix принято давать именам файлов статических библиотек суффикс .a, с расширением означает архив. Также принято давать любой библиотеке имя, начинающееся с lib.

```
$ gcc -g -c bc $ ar
rc lib88.a bo
```

Команда ar здесь создает библиотеку lib88.a из любых функций, которые она находит в файле bo. Затем вы можете скомпилировать свою основную программу:

```
$ gcc -g ac -l88 -Lz
```

Опция `-l` здесь является сокращением и имеет тот же эффект, что и

```
$ gcc -g ac lib88.a -Lz
```

что предписывает GCC сообщить LD, что ему необходимо найти функции в библиотеке `lib88.a` (или в динамическом варианте, как вы увидите ниже).

Опция `-L` указывает GCC указать LD искать в каталогах, отличных от текущего (и каталогов поиска по умолчанию), при поиске ваших функций. В этом случае она говорит, что `z` — это такой каталог.

Недостатком этого подхода является то, что если много программ используют одну и ту же библиотеку, каждая из них будет содержать отдельные копии на диске, которые занимают место. Эта проблема решается (за счет небольшого дополнительного времени загрузки) с помощью динамических библиотек.

В этом примере вы бы использовали GCC напрямую для создания динамической библиотеки, а не использовали бы `ar`. В `z` вы бы запустили

```
$ gcc -fPIC -c bc $ gcc  
-shared -o lib88.so bo
```

Это создает динамическую библиотеку `lib88.so`. (В Unix принято использовать суффикс `.so`, общий объект, возможно, с последующим номером версии, для именования динамических библиотек.) Создайте ссылку на него, как вы делали для статического случая:

```
$ gcc -g ac -l88 -Lz
```

Однако теперь это работает немного по-другому. Тогда как в статическом случае функции, вызываемые из библиотеки, стали бы частью нашего исполняемого файла `a.out`, теперь `a.out` будет просто содержать нотацию о том, что эта программа использует библиотеку `lib88.so`. И что важно, эта нотация даже не будет указывать, где находится эта библиотека. Единственной причиной, по которой GCC (опять же, на самом деле LD) хотел заглянуть в `lib88.so` во время компиляции, было получение информации о библиотеке, которая ему нужна для ссылки.

Сама ссылка будет выполнена во время выполнения. Операционная система будет искать `lib88.so`, а затем свяжет его с вашей программой. Это поднимает вопрос о том, где ОС выполняет этот поиск.

Прежде всего, давайте воспользуемся командой `ldd`, чтобы проверить, какие библиотеки нужны программе, и где ОС их находит, если находит:

```
$ ldd a.out  
lib88.so => не найден  
libc.so.6 => /lib/tls/libc.so.6 (0x006cd000) /lib/ld-linux.so.2  
(0x006b0000)
```

Программе нужна библиотека С, которую она находит в каталоге /lib/tls, но ОС не может найти lib88.so. Последний находится в каталоге /Debug/z, но этот каталог не является частью обычного пути поиска ОС.

Одним из способов исправить это является добавление /Debug/z к этому пути поиска:

```
% setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/Debug/z
```

Если вы хотите добавить несколько каталогов, соедините их имена вместе, используя двоеточия в качестве разделителей. (Это для оболочки С или оболочки ТС.) Для bash выполните команды

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/Debug/z $  
экспорт LD_LIBRARY_PATH
```

Давайте убедимся, что это работает:

```
$ ldd a.out  
lib88.so => /Debug/z/lib88.so (0xf6ffe000) libc.so.6 => /  
lib/tls/libc.so.6 (0x006cd000) /lib/ld-linux.so.2 (0x006b0000)
```

Существуют и другие подходы, но они выходят за рамки данной книги.

6.1.2.2 Использование библиотек в программном

обеспечении с открытым исходным кодом Программное обеспечение с открытым исходным кодом стало довольно популярным, особенно среди пользователей Linux. Однако иногда возникает проблема, заключающаяся в том, что скрипт сборки, обычно называемый `configure`, который поставляется с исходным кодом, не может найти некоторые необходимые библиотеки. Попытки решить проблему путем установки переменной среды `LD_LIBRARY_PATH` могут потерпеть неудачу. Поскольку это относится к теме «отсутствующие библиотеки», обсуждаемой здесь, и обычно не документируется в исходных пакетах, краткая заметка об этом может быть полезной.

Часто корень проблемы кроется в программе, вызываемой конфигурацией. с именем `pkgconfig`. Последний будет извлекать информацию о библиотеках из определенных файлов метаданных, которые имеют суффикс .pc, где префиксом является имя библиотеки. Например, файл `libgcj.pc` будет содержать местоположение файлов библиотеки `libgcj.so*`.

Каталог по умолчанию, в котором `pkgconfig` ищет файлы .pc, зависит от расположение самого `pkgconfig`. Например, если программа находится в `/usr/bin`, она будет искать `/usr/lib`. Этого будет недостаточно, если нужная библиотека — `/usr/local/lib`. Чтобы исправить эту проблему, задайте переменную окружения `PKG_CONFIG_PATH`. В оболочке С или ТС вам следует ввести команду оболочки

```
% setenv PKG_CONFIG_PATH /usr/lib/pkgconfig:/usr/local/lib/pkgconfig
```

6.2 Отладка программ с графическим интерфейсом

В наши дни пользователи привыкли, что их прикладные программы поставляются с графическими пользовательскими интерфейсами (GUI). Конечно, это всего лишь программы, поэтому общие принципы отладки применяются, но особые соображения все же вступают в игру.

Программирование GUI в основном состоит из вызовов библиотеки для выполнения различных операций на экране. Существует множество таких библиотек, которые широко используются. Очевидно, мы не можем охватить их все, и в любом случае принципы схожи.

Соответственно, мы выбрали самый простой пример — библиотеку *curses*. Она настолько проста, что многие люди вообще не считут ее GUI — один студент назвал ее «текстовым GUI», — но она передает суть.

6.2.1 Отладка Проклятия Программы

Библиотека *curses* позволяет программисту писать код, который будет перемещать курсор по экрану, менять цвета символов или заменять их для обратного воспроизведения видео, вставлять и удалять текст и т. д.

Например, текстовые редакторы, такие как Vim и Emacs, запрограммированы на *curses*. В Vim нажатие клавиши *j* переместит курсор на одну строку вниз.

Ввод *dd* приведет к тому, что текущая строка будет стерта, строки под ней сдвинутся на одну строку вверх, а строки над ней останутся неизменными. Эти действия достигаются вызовами функций в библиотеке *curses*.

Чтобы использовать *curses*, вы должны включить этот оператор в свой исходный код

```
#include <curses.h>
```

и вам необходимо подключить библиотеку *curses*:

```
gcc -g исходный_файл.c -lcurses
```

Давайте возьмем код ниже в качестве примера. Он запускает команду Unix *ps ax* для вывода списка всех процессов. В любой момент времени строка, на которой в данный момент находится курсор, будет выделена. Вы можете перемещать курсор вверх и вниз, нажимая клавиши *u* и *d* и т. д. Полный список команд смотрите в комментариях в коде.

Не волнуйтесь, если вы раньше не использовали *curses*, так как в комментариях рассказывается, что делает эта библиотека.

```
// psax.c; иллюстрация библиотеки curses
```

```
// прочтайте этот код «сверху вниз»: сначала эти комментарии и глобальные // переменные,
затем main(), затем функции, вызываемые main()
```

```
// запускает команду оболочки 'ps ax' и сохраняет последние строки ее вывода, //
столько, сколько поместится в окне; позволяет пользователю перемещаться вверх и вниз
```

```

// в окне, с возможностью завершить любой процесс, // который в данный момент
выделен

// использование: psax

// пользовательские команды:

//      'u': переместить выделение на строку
//      вверх 'd': переместить выделение на
//      строку вниз 'k': завершить процесс в текущей выделенной
//      строке 'l': повторно запустить 'ps ax' для
//      обновления 'q': выйти

// возможные расширения: разрешение прокрутки, чтобы пользователь мог // просмотреть
весь вывод 'ps ax', а не только последние строки; разрешение // переноса для длинных
строк; запрос подтверждения пользователя перед завершением // процесса

#define MAXROW 1000
#define MAXCOL 500

#include <curses.h> // обязательно

OKHO *scrn; // будет указывать на объект окна curses

char cmdoutlines[MAXROW][MAXCOL]; // вывод 'ps ax' (лучше использовать // malloc()) int
ncmdlines, //
количество строк в cmdoutlines nwinlines, // количество строк,
которые наш вывод "ps ax" занимает в окне // xterm (или эквиваленте)

winrow, // текущая позиция строки на экране
cmdstartrow, // индекс первой строки в cmdoutlines для отображения cmdlastrow; //
индекс последней строки в cmdoutlines для отображения

// переписывает строку в winrow жирным шрифтом
выделять() {

    int clinenum;
    attron(A_BOLD); // этот вызов библиотеки curses говорит, что все, что мы // напишем с
        этого момента (пока мы не скажем иное), // будет выделено
        жирным шрифтом

    // нам нужно будет переписать строку cmdoutlines, которая в данный момент
    отображается // на строке winrow на экране, чтобы получить жирный шрифт
    clinenum = cmdstartrow + winrow;
    mvaddstr(winrow,0,cmdoutlines[clinenum]);
   attroff(A_BOLD); // OK, оставляем жирный режим
refresh(); // отображаем изменения на экране
}

```

```

}

// запускает "ps ax" и сохраняет вывод в cmdoutlines runpsax() {

FILE *p; char ln[MAXCOL]; int row,tmp; p = popen("ps
ax","r"); // открыть конвейер UNIX (позволяет одной программе читать // вывод другой, как
если бы это был файл) for (row = 0; row < MAXROW; row+
+) { tmp = fgets(ln,MAXCOL,p); // прочитать одну
строку из конвейера if (tmp == NULL) break; // если конец конвейера, break //
не хотим, чтобы сохраненная строка превышала ширину экрана,
которую библиотека curses // предоставляет нам в переменной COLS, поэтому
обрезаем // до максимум COLS символов

strncpy(cmdoutlines[строка],ln,COLS);
cmdoutlines[строка][MAXCOL-1] = 0;

} ncmlines = row;
close(p); // закрыть канал
}

// отображает последнюю часть вывода команды (столько, сколько помещается на
экране)
showlastpart() {
int row;
clear(); // вызов очистки экрана curses // подготовка
к отрисовке (последней части) вывода 'ps ax' на экране; // два случая, в зависимости от того,
больше ли вывода, чем строк на экране; // во-первых, случай, когда весь вывод помещается на
одном экране: if (ncmlines <= LINES) { // LINES — это int, поддерживаемый библиотекой
curses, // равный количеству строк на экране

cmdstartrow = 0;
nwinlines = ncmlines;

} else { // теперь случай, когда вывод больше одного экрана cmdstartrow = ncmlines -
LINES; nwinlines = LINES;

} cmdlastrow = cmdstartrow + nwinlines - 1; // теперь
рисуем строки на экране for (row = cmdstartrow,
winrow = 0; row <= cmdlastrow; row++,winrow++) mvaddstr(winrow,0,cmdoutlines[row]); //
вызывает curses для перемещения в указанную позицию и // рисования там строки

refresh(); // теперь заставим изменения действительно появиться на экране, //
используя этот вызов библиотеки curses
}

```

```

// выделить последнюю строку
winrow--;
highlight();
}

// перемещает курсор вверх/вниз на одну
строку updown(int
inc) {
    int tmp = winrow + inc; //
игнорировать попытки выйти за край экрана if (tmp >= 0 && tmp <
LINES) {
        // переписать текущую строку перед перемещением; поскольку наш текущий шрифт //
        не жирный (на самом деле A_NORMAL), эффект заключается в том, чтобы снять
        выделение // этой строки
        mvaddstr(winrow,0,cmdoutlines[winrow]); // выделяем
        строку, на которую переходим winrow = tmp;
        highlight();

    }
}

// запустить/перезапустить "ps
ax" rerun()
{
    runpsax();
    showlastpart();
}

// завершает выделенный процесс prockill()
{

    char *pid; //
    strtok() из библиотеки C; см. страницу руководства pid =
    strtok(cmdoutlines[cmdstartrow+winrow]," "); kill(atoi(pid),9); // это
    системный вызов UNIX для отправки сигнала 9, // сигнала kill, данному процессу

    перезапустить();
}

основной()
{
    char c; //
    настройка окна; следующие 3 строки — вызовы библиотеки curses, стандартная //
    последовательность инициализации для программ curses
    scrn = initscr();
    noecho(); // не отображать нажатия клавиш
    cbreak(); // ввод с клавиатуры действителен немедленно, а не после нажатия Enter
}

```

```

// запустить 'ps ax' и обработать вывод
runpsax();
// отобразить в окне
показатьпоследнюючасть();
// цикл пользовательских команд
в то время как (1) {
    // получить команду пользователя
    c = getch();
    если (c == 'u') вверхвниз(-1);
    иначе если (c == 'd') вверхвниз(1);
    иначе если (c == 'r') перезапустить();
    иначе если (c == 'k') prockill();
    иначе break; // выход
}
// восстановить исходные настройки
endwin();
}

```

Запустив программу, вы увидите, что изображение выглядит нормально, но когда вы нажимаете клавишу **u**, чтобы переместить курсор на строку вверх, это работает неправильно, как вы видите на рисунке 6-1.

Процесс	Приоритет	Время выполнения	Команды
5912 ?	S	0:00	/usr/lib/nautilus-cd-burner/mapping-daemon
5916 ?	S	0:00	klauncher [kdeinit]
5922 ?	S	0:00	gnome-pty-helper
5923 pts/0	Ss	0:00	-csh
5926 ?	S	0:00	kded [kdeinit]
5949 ?	S	0:00	/usr/lib/fast-user-switch-applet/fast-user-switch-app
5951 ?	SL	0:00	/usr/bin/python /usr/lib/desktop-applet/desktop-apple
5953 ?	SL	0:00	/usr/lib/gnome-applets/mixer_applet2 --oaf-activate-i
5962 pts/0	S+	0:01	ssh laura.cs.ucdavis.edu -l matloff
5964 ?	S	0:01	/usr/lib/notification-daemon/notification-daemon
5966 pts/1	Ss+	0:00	-csh
6498 ?	S	0:00	/bin/sh /usr/bin/firefox
6518 ?	S	0:00	/bin/sh /usr/lib/firefox/run-mozilla.sh /usr/lib/fire
6514 ?	SL	0:52	/usr/lib/firefox/firefox-bin
6545 pts/2	Ss+	0:00	-csh
6971 pts/4	Ss	0:00	-csh
7114 ?	S	0:00	knotify [kdeinit]
7117 ?	S	0:03	/usr/bin/artsd -F 10 -S 4096 -s 60 -m artsmessage -l
7118 ?	S	0:00	kio_file [kdeinit] file
7146 pts/4	S	0:00	qiv 06-f1g61.jpg
7149 pts/1	S	0:07	xpdf my6.pdf
7162 pts/4	S+	0:00	psax
2270 ?	S<	0:00	[ata/0]
2271 ?	S<	0:00	[ata/1]

Рисунок 6-1: Окно терминала

Вывод **ps ax** идет в порядке возрастания номера процесса, но внезапно вы видите, что процесс 2270 отображается после 7162. Давайте проследим ошибку.

Программа **curses** — мечта автора книги по отладке, потому что она заставляет программиста использовать отладочный инструмент. Программист не может использовать **printf()** вызовы или операторы **cout** для вывода отладочной информации, потому что это вывод будет смешан с выводом самой программы, что приведет к безнадежным хаос.

6.2.1.1 Использование

GDB Итак, запустите GDB, но есть еще одна дополнительная вещь, которую мы должны сделать, связанная с этим последним пунктом. Мы должны указать GDB, чтобы программа выполнялась в другом окне терминала, а не в том, в котором запущен GDB. Мы можем сделать это с помощью команды `tty` GDB . Сначала мы переходим в другое окно, в котором будет выполняться ввод-вывод программы, и запускаем там команду `tty Unix` , чтобы определить идентификатор этого окна. В этом случае вывод этой команды говорит нам, что окно имеет номер терминала `/dev/pts/8`, поэтому мы вводим

```
(gdb) терминал /dev/pts/8
```

в окне GDB. С этого момента весь ввод с клавиатуры и вывод на экран для программы будет находиться в окне выполнения.

И последнее, прежде чем мы начнем: мы должны ввести что-то вроде

```
сон 10000
```

в окне выполнения, так что наш ввод с клавиатуры в этом окне будет поступать в программу, а не в оболочку.

ПРИМЕЧАНИЕ Существуют и другие способы решения проблемы разделения вывода GDB и вывода программы. Например, мы могли бы сначала запустить выполнение программы, а затем запустить GDB в другом окне, прикрепив его к работающей программе.

Далее устанавливаем точку останова в начале функции `updown()`, так как ошибка возникает при попытке переместить курсор вверх. Затем вводим `r`, и программа начнет выполнятся в окне выполнения. Нажмите клавишу `U` в этом окне, и GDB остановится в точке останова.

```
(gdb) r
Запуск программы: /Debug/psax
Отсоединение после открытия от дочернего процесса 3840.
```

```
Точка останова 1, вверх-вниз (inc=-1) в psax.c:103 { int
103      tmp = winrow + inc;
```

Сначала давайте удостоверимся, что переменная `tmp` имеет правильное значение.

```
(гдб) н
105      если (tmp >= 0 && tmp < СТРОКИ) { (gdb)
p tmp $2 = 22
(gdb) р
СТРОКИ $3 = 24
```

Переменная `winrow` показывает текущее местоположение курсора в окне. Это местоположение должно быть в самом конце окна. `LINES` имеет значение 24, поэтому `winrow` должно быть 23, так как нумерация начинается с 0. С `inc`

равно -1 (поскольку мы перемещали курсор вверх, а не вниз), показанное здесь значение tmp , 22, подтверждается.

Теперь перейдем к следующей строке.

```
(гдб) н
109          mvaddstr(winrow,0,cmdoutlines[winrow]); (gdb) p
cmdoutlines[winrow] $4 = 2270 ?
"                                     SS      0:00 nifd -n\n", '0' <повторяется 464 раза>
```

Конечно, вот она, строка для процесса 2270. Мы быстро понимаем, что линия

```
mvaddstr(winrow,0,cmdoutlines[winrow]);
```

в исходном коде должно быть

```
mvaddstr(winrow,0,cmdoutlines[cmdstartrow+winrow]);
```

Как только мы это исправим, программа заработает нормально.

Когда закончим, нажмите CTRL-C в окне выполнения, чтобы завершить команду sleep и снова сделать оболочку доступной для использования.

Обратите внимание, что если что-то пойдет не так и программа завершится преждевременно, то в этом окне выполнения могут остаться некоторые нестандартные настройки терминала, например, режим cbreak. Чтобы исправить это, перейдите в это окно и нажмите CTRL-J, затем введите слово reset, затем снова нажмите CTRL-J .

6.2.1.2 Использование

DDD Что насчет DDD? Опять же, вам понадобится отдельное окно для выполнения программы. Организуйте это, выбрав Вид | Окно выполнения, и DDD откроет окно выполнения. Обратите внимание, что вам не нужно вводить команду sleep в этом окне, так как DDD сделает это за вас. Теперь экран будет выглядеть так, как показано на рисунке 6-2.

Установите точки останова как обычно, но не забудьте ввести входные данные программы окно исполнения.

6.2.1.3 Использование

Eclipse Сначала обратите внимание, что при сборке проекта вам необходимо указать Eclipse использовать флаг -lcurses в makefile, процедура которого была показана в Главе 5.

Здесь вам также понадобится отдельное окно выполнения. Вы можете сделать это, когда настроите диалог отладки. После настройки диалога запуска, как обычно, и выбора Run | Open Debug Dialog, мы пойдем немного другим путем, чем тот, который мы делали до сих пор. Обратите внимание на рисунок 6-3, что в дополнение к обычному выбору C/C++ Local Application, есть также опция C/C++ At-tach to Local Application. Последнее означает, что вы хотите, чтобы Eclipse использовал способность GDB присоединяться к уже запущенному процессу (обсуждается

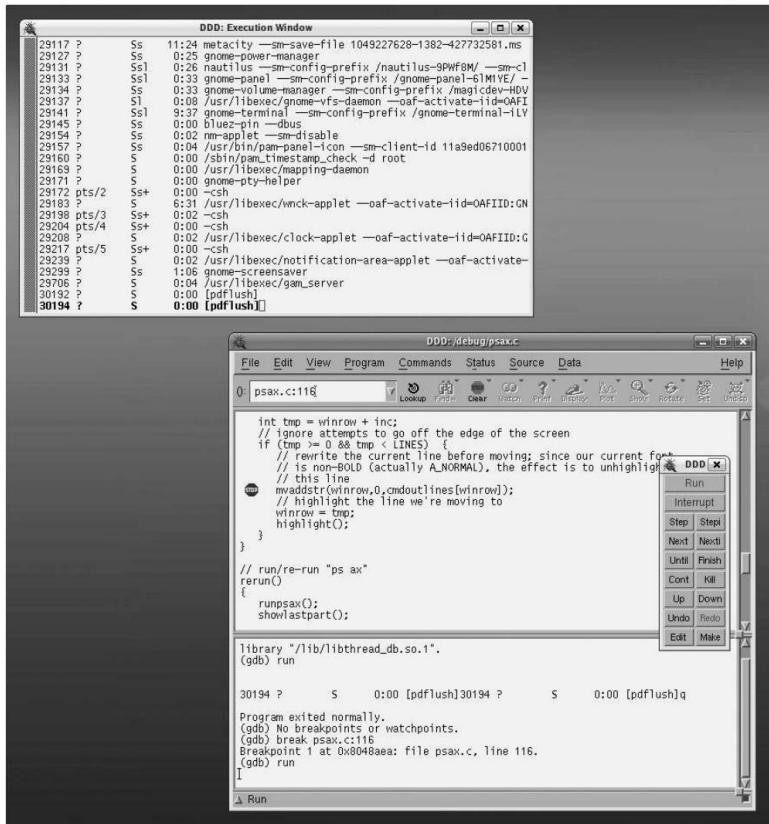


Рисунок 2: Прикрепление к программе обкатки

ДДД

в Главе 5). Щелкните правой кнопкой мыши C/C++ Присоединить к локальному приложению, выберите Создать, и продолжайте как прежде.

Когда вы начинаете реальный отладочный прогон, сначала запустите свою программу в отдельном окне оболочки. (Не забывайте, что программа, вероятно, находится в вашем Рабочая папка Eclipse.) Затем выберите Выполнить | Открыть диалоговое окно отладки. обычно это происходит при первом запуске отладки; в этом случае Eclipse появится окно со списком процессов и предложением выбрать тот, который нужно который вы хотите присоединить к GDB. Это показано на рисунке 6-4, который показывает, что Ваш процесс psax имеет идентификатор 12319 (обратите внимание на программу, запущенную в другом окне, частично скрытом здесь). Щелкните этот процесс, затем щелкните OK, что приведет к ситуации, изображенная на рисунке 6-5.

На этом рисунке вы можете видеть, что мы остановились во время системного вызова. Eclipse сообщает, что исходного кода для текущей инструкции нет, но это ожидаемо и не проблема. На самом деле, это хорошее время, чтобы Установите точки останова в исходном файле psax.c. Сделайте это, а затем нажмите кнопку Resume значок. Eclipse будет работать до тех пор, пока не достигнет первой точки останова, а затем вы сможете выполнить отладку как обычно.

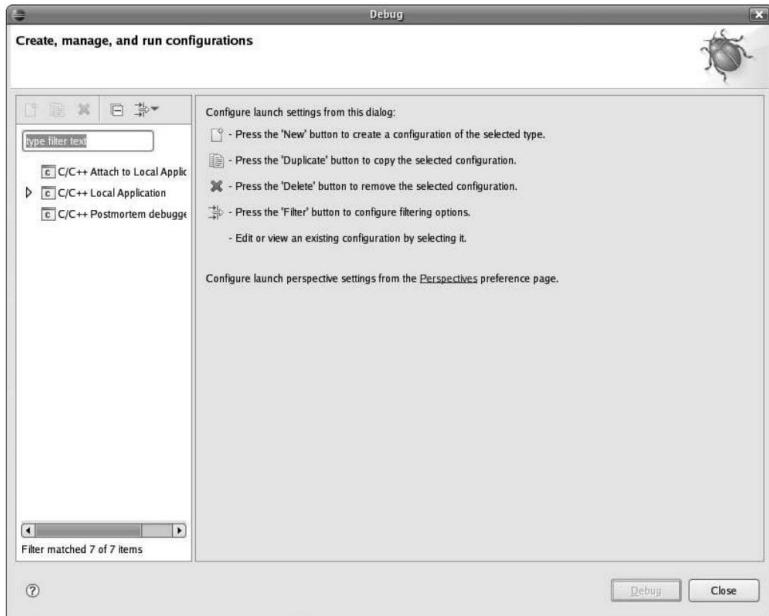
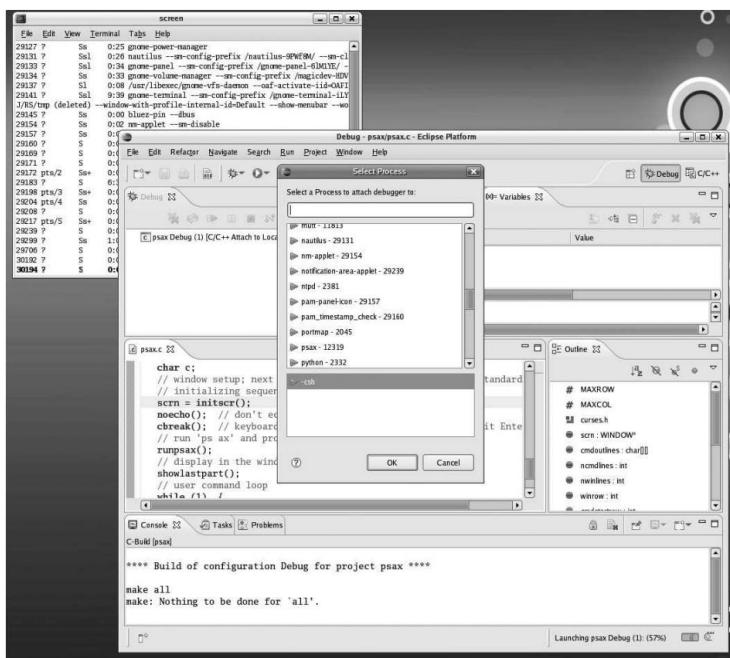
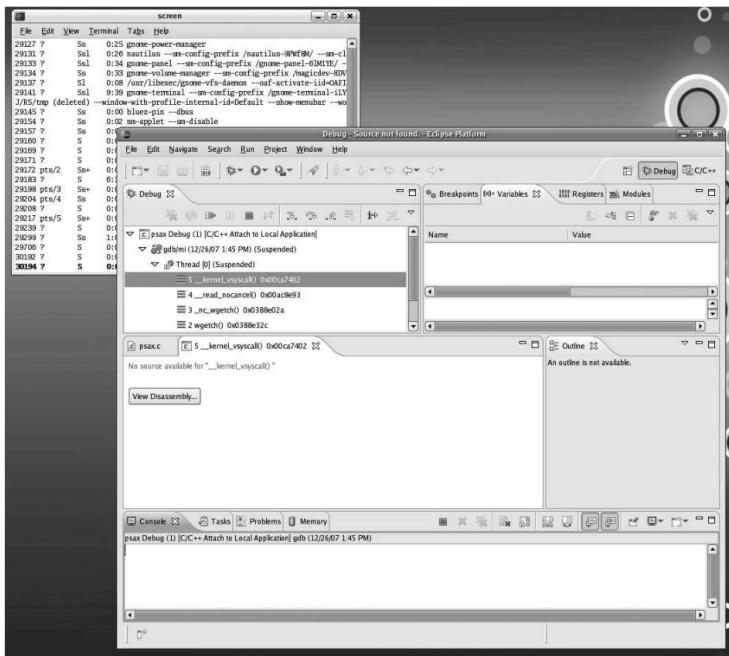


Рисунок 3: Присоединение к запущенной программе Затмение



Фигура 4: Выбор процесса В прикреплять GDB



Фигура 6-5: Остановил ядро

7

ДРУГИЕ ИНСТРУМЕНТЫ



Приобретение навыков отладки кода не заканчивается на изучении отладчика вроде GDB — оно начинается там. Есть множество других инструментов, как бесплатных, так и коммерческих, которые также помогают предотвращать, обнаруживать и устранять ошибки в коде. Опытный программист сохраняет несколько их в своем арсенале трюков, понимает, что каждый подходит для и распознает, когда использовать один из них чтобы сэкономить время и усилия в случае появления ошибок.

До сих пор мы сосредоточили свое внимание на использовании символьических отладчиков, но теперь мы хотели бы расширить наше освещение на другие аспекты отладки, включая защитное программирование. Эта глава посвящена некоторым инструментам и методам, отличные от GDB, которые могут оказаться полезными как для предотвращения возникновения ошибок, так и для их обнаружения и исправления, когда они возникнут они это делают.

7.1 Эффективное использование текстового редактора

Лучший метод отладки — не допускать ошибок программирования с самого начала.

! Простое использование редактора с поддержкой программирования — один из самых недооцененных аспектов «предварительной отладки».

Если вы тратите много времени на кодирование, мы настоятельно рекомендуем вам тщательно подумать о ваш выбор редактора и узнать как можно больше о редакторе, который вы будете использовать.

Есть две основные причины. Во-первых, стать профессионалом в

Мощный редактор сокращает время, необходимое для написания кода. Редакторы, которые имеют специализированные функции, такие как автоматический отступ, завершение слов и Глобальный поиск символов — это благо для программистов, и есть несколько

выбирайте из. Во-вторых, хороший редактор может помочь кодеру отловить определенные виды ошибок по мере написания кода. Вот о чем этот раздел

о.

Оба автора этой книги используют Vim для программирования, поэтому мы будем использовать именно его.

сосредоточиться на. Однако все популярные редакторы имеют похожие, если не идентичные, наборы

функций. Если Vim будет улучшен, чтобы предоставить полезную функцию, которой в настоящее время нет

Emacs, сообщество разработчиков Emacs быстро исправит ситуацию, и наоборот. Поэтому,

хотя мы и даем конкретику для Vim, большинство

То, о чем мы говорим, применимо и к другим прекрасным редакторам, таким как Emacs.

7.1.1 Синтаксис Выделение

Vim использует подсветку синтаксиса для отображения частей программного файла разными цветами или шрифтами, так что такие элементы кода, как ключевые слова, идентификаторы типов, локальные переменные и директивы препроцессора имеют свои собственные цвета и Шрифтовая схема. Редактор выбирает цветовую и шрифтовую схему, глядя на расширение имени файла, чтобы определить используемый вами язык. Например, если имя файла заканчивается на .pl (указывает на скрипт Perl), вхождения слова die (это имя функции Perl) подсвечиваются, тогда как если имя файла заканчивается на .c, то это не так.

Лучшим названием для подсветки синтаксиса было бы лексическое подсветка , потому что редактор обычно не анализирует синтаксис слишком подробно. Он не может вам сказать что вы указали неправильное количество аргументов или аргументов

Неправильный тип в вызове функции. Вместо этого он понимает только (например)

что такие слова, как bless и foreach, являются ключевыми словами Perl, a fmt и dimension являются ключевыми словами Фортрана и отображает их соответствующим образом.

Тем не менее, подсветка синтаксиса по-прежнему очень полезна для обнаружения простых, но легко сделать ошибки. Например, на нашем компьютере цвет по умолчанию для идентификаторы типа, такие как ключевые слова FILE или float в C, зеленые. Как только ваш глаз приучившись к шрифту и цветовой схеме, вы заметите несоответствие цветов, возникающее при неправильном написании названия шрифта, и автоматически исправите его ошибку, не проходя через ненужный цикл компиляции.

Пример использования подсветки синтаксиса для проверки ключевых слов, которые мы (авторы) наслаждаются происходит с makefiles. Ключевое слово patsubst очень полезно текстовая команда поиска и замены для makefiles. Одна из самых распространенных используется для генерации списка файлов .o из файлов .c исходного кода проекта:

```
ЦЕЛЬ = CoolApplication
OBJS = $(patsubst %.c, %.o, $(wildcard *.c))
```

```
$(ЦЕЛЬ): $(ОБЪЕКТЫ)
```

Один из авторов никогда не помнит, patsubst ли это, pathsubst, или patsub. Зная, что ключевые слова makefile отображаются светлым цветом (желтым), можете ли вы определить, какая версия строки, показанной ниже, неверна? Даже если вы не знаете, как писать make-файлы, одной подсветки синтаксиса будет достаточно. должно быть ясно!¹

```
OBJ = $(patsub %.c, %.o $(wildcard *.c))
OBJ = $(patsubst %.c, %.o $(wildcard *.c))
```

Рисунок 7-1: Подсветка синтаксиса находит ошибку в моем make-файле.

Более того, вот пример, где подсветка синтаксиса немного умнее. На рисунке ниже есть самая настоящая синтаксическая ошибка. Попробуйте найти, не слишком задумываясь, в чем (или хотя бы где) ошибка, основываясь на только цвета:

```
если (fp == NULL) {
    puts("Это был \"плохой\" указатель файла.");
    выход(1);
}
```

Рисунок 7-2: Синтаксис выделение показывает распространенная ошибка.

и вот еще одна иллюстрация похожей ошибки. Попробуйте дать цветовому руководству ваш взгляд на ошибку.

```
fprintf(fp, "аргумент %d равен \"%s\".\n", i, argv[i]);
printf("Я только что написал \"%s\", что является аргументом %d\n", argv[i], i);
```

Фигура 7-3: Подсветка синтаксиса раскрывает еще один распространенная ошибка.

Вы можете обнаружить, что некоторые цвета в схеме подсветки синтаксиса трудно читать. Если это так, вы можете отключить подсветку, введя следующее:

```
: синтаксис выключен
```

¹ Эту цифру нужно было преобразовать в оттенки серого для книги. На практике, выявление ошибки было бы еще проще.

Команда для повторного включения, конечно же,

: синтаксис включен

Лучшим вариантом было бы изменить файл синтаксиса, чтобы использовать другой, более подходящий цвет для этого типа ключевого слова, но это выходит за рамки нашего обсуждения.

7.1.2 Соответствие Скобки

Ошибки несбалансированных скобок чрезвычайно распространены и их очень трудно отловить. Рассмотрим следующий код:

```
mytype *myvar;
если ((myvar = (mytype *)malloc(sizeof(mytype))) == NULL) {
    выход(-1);
}
```

ПРИМЕЧАНИЕ. В этом разделе слово «скобки» относится к круглым скобкам, квадратным скобкам и фигурным скобкам (или фигурные скобки): (), [], {} соответственно.

Быстро: сбалансированы ли скобки? Вам когда-нибудь приходилось отслеживать несбалансированные скобки в длинных блоках кода со множеством условных операторов? Или пытались использовать TEX? (Мы содрогаемся, вспоминая некоторые из наших прошлых файлов LATEX с отсутствующими скобками!) Тогда вы должны согласиться с нами, что именно для этого и нужны компьютеры — чтобы избавить нас от такой утомительной работы! В Vim есть несколько замечательных функций, которые могут помочь.

- Всякий раз, когда вы вводите скобку на клавиатуре, опция Vim showmatch Vim мгновенно помещает курсор на парную скобку (если парная скобка существует и видна на экране). Вы даже можете контролировать, как долго курсор задерживается на парной скобке, установив переменную matchtime , которая определяет эту продолжительность в десятых долях секунды. • Ввод символа процента, когда курсор находится на скобке, переместит курсор на пару скобки. Эта команда — отличный способ отследить несбалансированные скобки.

- Когда вы помещаете курсор на скобку, Vim подсветит ее пару, как показано на рисунке ?? . Это также отличный способ отслеживать проблемы с несбалансированными скобками.

Параметр showmatch полезен, когда вы программируете, но в противном случае он может раздражать. Вы можете использовать автокоманды, чтобы задать этот параметр только при программировании. Например, чтобы задать showmatch только для сеансов редактирования с файлами исходного кода C/C++, вы можете поместить такие строки в свой файл .vimrc (см. файлы справки Vim для получения дополнительной информации):

2 Признайтесь. Вы ожидали, что они не будут балансировать. Мы не говорим, что вы были неправы, а то, что вам, вероятно, потребовалось больше времени, чем следовало, чтобы ответить на вопрос!

или BufNewFile,BufRead *.c set showmatch или
BufNewFile,BufRead *.cc set showmatch

Что делать, если в ваш код попала несбалансированная скобка или, что еще хуже, вам нужно отловить несбалансированные скобки в чьем-то спагетти-коде? Ранее упомянутая команда редактора % ищет сбалансированные символы группировки.

Например, если вы поместите курсор на левую квадратную скобку [и наберете % из командного режима, Vim переместит курсор на следующий символ]. Если вы поместите курсор на правую фигурную скобку } и вызовете %, Vim переместит курсор на предыдущий совпадающий символ { . Таким образом, вы можете проверить не только то, что любая заданная круглая, фигурная или квадратная скобка имеет соответствующего партнера по сопоставлению, но и то, что партнер совпадает семантически.

Вы даже можете определить другие соответствующие пары «скобок», например, разделители комментариев HTML <!-- и -->, используя команду Vim matchpair . Для получения дополнительной информации см. страницы справки Vim.

7.1.3 Вим И Makefile-ы

Утилита make управляет компиляцией и сборкой исполняемых файлов в системах Linux/Unix, и небольшое усилие, затраченное на изучение того, как ее использовать, может принести большие дивиденды программисту. Однако это также создает новые возможности для ошибок. Vim имеет несколько функций, которые могут действительно помочь в процессе отладки, если вы используете make. Рассмотрим следующий фрагмент makefile:

```
все: perror.o main.o gcc
      -o myprogram perror.o main.o

perror.o: perror.c perror.h
      gcc -c perror.c

main.o: main.c main.h
      gcc -c main.c
```

В этом makefile есть ошибка, но ее трудно заметить. make очень требователен к форматированию. Командная строка цели должна начинаться с символа табуляции, а не с пробелов. Если вы выполните команду set list из Vim, вы сразу увидите, в чем проблема:

```
все: perror.o main.o$ ^Igcc
      -o myprogram perror.o main.o$ $ perror.o:

perror.c perror.h$ ^Igcc -c perror.c$
      $ main.o: main.c

main.h$ gcc -c main.c$
```

В режиме списка Vim отображает непечатаемые символы. По умолчанию символ конца строки отображается как \$, а управляющие символы отображаются с символ вставки (^); таким образом, символ табуляции, то есть CTRL-I, отображается как ^I. Таким образом, вы можете отличить пробелы от табуляции, и ошибка становится очевидной: Командная строка для цели main.o make начинается с пробелов.

Вы можете управлять тем, что отображается, с помощью опции listchars в Vim .

Например, если вы хотите изменить символ конца строки на = вместо \$, вы можете использовать :set listchars=eol:=.

7.1.4 Makefiles и

Компилятор Предупреждения

Вызов make из Vim может быть очень удобен. Например, вместо вручную сохраните файл и введите make clean в другом окне, все, что вам нужно нужно сделать, это ввести :make clean из командного режима. (Убедитесь, что автозапись (Установлено так, чтобы Vim автоматически сохранял файл перед запуском команды make .) В общем, всякий раз, когда вы вводите

:приводить аргументы

из командного режима Vim запустит make и передаст ему аргументы .

Это становится еще лучше. Когда вы создаете свою программу в Vim, редактор фиксирует все сообщения, которые выдает компилятор. Он понимает синтаксис вывода GCC и знает, когда возникает предупреждение или ошибка компилятора. Давайте посмотрим на это в действии. Рассмотрим следующий код:

```
#include <stdio.h>

int main(пустота)
{
    printf("Было %d аргументов.\n", argc);

    если (argc > 5) тогда
        напечатайте *, «Сегодня вы, кажется, любите спорить»;
    конец_если

    возврат 0;
}
```

Листинг 7-1: main.c

Похоже, кто-то немного поработал над параллельным Фортраном и Си. кодирование! Предположим, вы в данный момент редактируете main.c и хотите собрать программу. Выполните команду :make из Vim и просмотрите все сообщения об ошибках (рисунок 7-4).

Теперь, если вы нажмете ENTER или пробел, вы вернетесь к редактированию программы, но с курсором, установленным на строке, которая сгенерировала первый предупреждение или ошибка (в данном случае сообщение о том, что argc не был объявлен) как показано на рисунке 7-5.

```
:!make 2>&1| tee /tmp/v243244/1
gcc -std=c99 -W -Wall main.c -o main
main.c: In function 'main':
main.c:12: error: 'argc' undeclared (first use in this function)
main.c:12: error: (Each undeclared identifier is reported only once
main.c:12: error: for each function it appears in.)
main.c:14: error: expected identifier before numeric constant
main.c:14: error: 'then' undeclared (first use in this function)
main.c:15: error: expected ';' before 'print'
main.c:15:18: warning: character constant too long for its type
main.c:16: error: 'end' undeclared (first use in this function)
main.c:16: error: expected ';' before 'if'
make: *** [main] Error 1
(3 of 12): error: 'argc' undeclared (first use in this function)
Press ENTER or type command to continue
```

Фигура 7-4: The ошибки сообщения

```
#include <stdio.h>

int main(void)
{
    printf("There were %d arguments.\n", argc);

    if (argc > 5) then
        print *, 'You seem argumentative today';
    end if

    return 0;
}
```

Рисунок7-5: The как курсор позиционируется первый в ошибке.

После исправления ошибки есть два способа перейти к следующей ошибке:

- Вы можете переделать программу, и Vim снова отобразит оставшиеся предупреждения и ошибки и переместит курсор на первое из них. Это имеет смысл, если время сборки незначительно, особенно если вы отображаете один нажатие клавиши для сборки программы, например:

```
au BufNewFile,BufRead *.c map <F1> :make<CR>
```

- Вы также можете использовать :спехт, который отображает следующую ошибку или предупреждение. Аналогично, :сprevious отображает последнюю ошибку или предупреждение, а :cc отображает текущую ошибку или предупреждение. Все три команды удобно перемещают курсор к месту «активной» ошибки или предупреждения.

7.1.5 Финал Мысли На Текст Редактор Как ИДЕ

Освоение выбранного вами редактора настолько очевидно, что часто пренебрегают, но это действительно первый шаг в обучении программированию в определенной среде. Рискуя преувеличить, редакторы для программистов — то же, что музыкальные инструменты для музыкантов. Даже самые креативные композиторы должны знать основы игры на инструменте, чтобы реализовать свои идеи, чтобы другие люди могли извлечь из них пользу. Обучение использованию ваш редактор в полной мере позволяет вам программировать быстрее, сообразите

более эффективно анализировать чужой код и сокращать количество циклов компиляции, которые необходимо выполнять при отладке кода.

Если вы используете Vim, мы рекомендуем Vi IMproved—Vim Стива Уэллина (новый) Riders, 2001). Книга подробная и хорошо написана. (К сожалению, она была написана для Vim 6.0, а Vim 7.0 и более поздние функции, такие как сворачивание, не рассматриваются.) Наша цель здесь была просто дать представление о том, что Vim может сделать для программиста, но книга Стива — отличный источник для изучения деталей.

Vim имеет много функций, которые авторы считают непристойно полезными. Например-
ple, мы бы хотели охватить

- Использование K для поиска функций на страницах руководства
- Поиск объявлений переменных с помощью gd и gD
- Переход к определениям макросов с помощью [^D и]^D
- Отображение определений макросов с помощью]d,]D, [D и]D
- Разделение окна для одновременного просмотра файлов .c и .h с целью проверки прототипов
- ... и многое другое

Но это книга по отладке, а не по Vim, и нам нужно вернуться к обсуждение дополнительных программных инструментов.

7.2 Эффективное использование компилятора

Если ваш редактор — ваше первое оружие в битве с ошибками, то ваш компилятор — ваше второе. Все компиляторы имеют возможность сканировать код и находить распространенные ошибки, но обычно вам приходится включать эту проверку ошибок, вызывая соответствующую опцию.

Многие из опций предупреждений компилятора, например, ключ GCC -Wtraditional , вероятно, излишни, за исключением особых ситуаций. Однако даже не думайте использовать GCC без использования -Wall каждый раз. Например, одна из самых распространенных ошибок, которые допускают начинающие программисты на C, проиллюстрирована следующим утверждением:

```
if (a = b)
    printf("Равенство для всех!\n");
```

Это допустимый код C, и GCC с радостью его скомпилирует. Переменной a присваивается значение b, и это значение используется в условном операторе. Однако это почти наверняка не то, что имел в виду программист. Используя ключи -Wall GCC , вы, по крайней мере, получите предупреждение, предупреждающее вас о том, что этот код может быть ошибкой:

```
$ gcc try.c $ gcc
-Wall try.c try.c: В
функции `main': try.c:8:
предупреждение: рекомендуется заключать в скобки значение присваивания, используемое в качестве истинного значе
```

GCC предлагает заключить в скобки присваивание `a=b` перед использованием это как истинное значение, так же, как вы это делаете, когда назначаете значение и выполнить сравнение: `if ((fp = fopen("myfile", "w")) == NULL)`. GCC по сути спрашивает: «Вы уверены, что хотите здесь присвоить `a=b`, а не тест на равенство `a == b?`»

Вам всегда следует использовать возможности проверки ошибок вашего компилятора, и если Вы преподаете программирование, вы должны требовать от своих студентов также их использования, для того, чтобы привить хорошие привычки. Пользователи GCC должны всегда использовать `-Wall`, даже для самая маленькая программа «Hello, world!» Мы обнаружили, что разумно использовать `-Wmissing-prototypes` и `-Wmissing-declarations`, а также. Действительно, если у вас есть 10 минут свободного времени, просмотр страницы руководства GCC и чтение компилятора раздел предупреждений — отличный способ провести время, особенно если вы собираетесь программировать в Unix в какой-либо значительной степени.

7.3 Сообщения об ошибках в C

Отчет об ошибках в языке C осуществляется с помощью старого механизма с именем `errno`. Хотя `errno` показывает свой возраст и имеет некоторые недостатки, он в целом выполняет свою работу. Вы можете задаться вопросом, зачем вам нужен отчет об ошибках механизм вообще, так как большинство функций C имеют удобное возвращаемое значение, которое сообщает был ли вызов успешным или нет. Ответ в том, что возвращаемое значение может предупредить вас, что функция не сделала то, что вы хотели, но это может быть или может быть не скажу почему. Чтобы сделать это более конкретным, рассмотрим этот фрагмент кода:

```
ФАЙЛ *fp
fp = fopen("myfile.dat", "r");
retval = fwrite(&data, sizeof(DataStruct), 1, fp);
```

Предположим, вы проверяете `retval` и обнаруживаете, что он равен нулю. Из страницы руководства: вы видите, что `fwrite()` должна возвращать количество записанных элементов (не байтов или символов), поэтому `retval` должен быть равен 1. Сколько различных способов может `fwrite()` ошибиться? Много! Для начала, файловая система может быть заполнена, или у вас может не быть разрешения на запись в файл. В этом случае, однако, есть ошибка в коде что приводит к сбою `fwrite()`. Можете ли вы его найти?³ Система сообщений об ошибках, например `errno` может предоставить диагностическую информацию, которая поможет вам выяснить, что произошло в таких случаях. (Операционная система также может сообщать об определенных ошибках.)

7.3.1 ошибка С использованием

Системные и библиотечные вызовы, которые завершаются неудачей, обычно устанавливают глобально определенное целое число. Переменная с именем `errno`. В большинстве систем GNU/Linux `errno` объявляется в `/usr/include/errno.h`, поэтому, включив этот заголовочный файл, вам не придется деобъявлять `extern int errno` в вашем собственном коде.

³ Мы открыли файл в режиме чтения, а затем попытались записать в него данные.

Когда системный или библиотечный вызов завершается неудачей, он устанавливает `errno` в значение, которое указывает тип сбоя. Вам решать, проверить ли значение `errno` и предпринять соответствующие действия. Рассмотрим следующий код:

```
#include <stdio.h>
#include <errno.h>
#include <math.h>

int main(пустота)
{
    двойная проблема = exp(1000.0);
    если (ошибка) {
        printf("проблема: %f (ошибка: %d)\n", проблема, ошибка);
        выход(-1);
    }

    возврат 0;
}
```

Листинг 7-2: двойная неприятность.c

В нашей системе `exp(1000.0)` больше, чем может хранить тип `double`, поэтому присваивание приводит к переполнению с плавающей точкой. Из вывода вы видите что значение `errno` 34 указывает на ошибку переполнения числа с плавающей точкой:

```
$ ./г.выход
проблема: инф (errno: 34)
```

Это в значительной степени иллюстрирует, как работает `errno`. По соглашению, когда библиотечная функция или системный вызов терпят неудачу, они устанавливают `errno` в значение, которое описывает, почему вызов не удался. Вы только что увидели, что значение 34 означает результат `exp(1000.0)` не может быть представлено как `double`, и есть много других кодов, которые указывают на underflow, проблемы с правами доступа, файл не найден и другие состояния ошибок. Однако, прежде чем вы начнете использовать `errno` в своих программах, есть некоторые вопросы, о которых вам следует знать.

Во-первых, код, использующий `errno`, может быть не полностью переносимым. Например, стандарт ISO C определяет только несколько кодов ошибок, а стандарт POSIX определяет гораздо больше. Вы можете увидеть, какие коды ошибок определены which standards in `errno man page`. Более того, стандарты не определяют числовые значения, например 34, для кодов ошибок. Они предписывают символическую ошибку коды, которые являются макроконстантами, имена которых начинаются с `E` и которые определены в заголовочном файле `errno` (или в файлах, включенных в заголовок `errno`). Единственное, что в их ценностях остается неизменным на всех платформах, это то, что они не равны нулю. Таким образом, вы не можете предполагать, что определенное значение всегда индицирует

указывает на то же самое состояние ошибки.⁴ Вы всегда должны использовать символьические имена для ссылки на значения errno .

В дополнение к значениям ошибок ISO и POSIX , конкретные реализации или библиотеки C , например, GNU glibc , могут определять даже больше значений errno . В GNU/Linux раздел errno страницы информации libc5 является каноническим источником всех доступных значений errno на этой платформе: ISO, POSIX и glibc . Вот некоторые определения кодов ошибок, которые мы вытащили из /usr/include/asm/errno.h с машины GNU/Linux:

#определить ЭПАЙП	32 /* Сломанная труба */
#определить EDOM	33 /* Математический аргумент вне области действия функции */
#определить ERANGE	34 /* Математический результат непредставим */
#определить EDEADLK	35 /* Произойдет взаимоблокировка ресурсов */
#define ENAMETOOLONG 36 /* Имя файла слишком длинное */ #define ENOLCK #define ENOSYS	36 /* Нет доступных блокировок записей */ 38 /* Функция не реализована */

Далее, есть несколько важных фактов, которые следует запомнить о том, как используется errno . errno может быть установлена любой библиотечной функцией или системным вызовом, независимо от того, завершается ли он успешно или нет! Поскольку даже успешные вызовы функций могут устанавливать errno , вы не можете полагаться на errno , чтобы узнать , произошла ли ошибка. Вы можете полагаться только на него, чтобы узнать , почему произошла ошибка. Поэтому самый безопасный способ использования errno следующий:⁶

1. Выполнить вызов библиотечной или системной функции.
2. Используйте возвращаемое функцией значение, чтобы определить, произошла ли ошибка. произошло.
3. Если произошла ошибка, используйте errno , чтобы определить причину.

В псевдокоде:

```
retval = системный вызов();

если (retval указывает на ошибку) { exam_errno();
    take_action();
}
```

Это приводит нас к тап-страницам. Предположим, вы пишете код и хотите добавить проверку ошибок после вызова ptrace(). Шаг два говорит об использовании возвращаемого значения ptrace() для определения того, произошла ли ошибка. Если вы похожи на нас,

⁴ Например, некоторые системы различают EWOULDBLOCK и EAGAIN , но GNU/Linux не делает этого.

нет.⁵ В дополнение к страницам информации libc , вы можете просмотреть файлы заголовков вашей системы, чтобы изучить значения errno . Это не только безопасно и естественно, но и на самом деле поощряемая практика!

⁶ Использование errno в листинге 7-2 не было хорошей практикой.

вы не запомнили возвращаемые значения `ptrace()`. Что можно сделать?

На каждой странице руководства есть раздел «Возвращаемое значение». Вы можете быстро перейти к нему, введя имя функции `man` и выполнив поиск возвращаемого значения строки .

Хотя у `errno` есть некоторые недостатки, есть и хорошие новости.

В библиотеке GNU C ведется обширная работа по сохранению `errno` при входе в функцию и последующему восстановлению его до исходного значения, если вызов функции успешен. Похоже, что glibc изо всех сил старается не перезаписывать `errno` при успешных вызовах функций. Однако мир не GNU (пока), поэтому переносимый код не должен полагаться на этот факт.

Кроме того, хотя и просматривая документацию каждый раз, когда вы хотите увидеть что означает конкретный код ошибки, становится утомительным, есть две функции, которые облегчают интерпретацию кодов ошибок: `perror()` и `strerror()`. Они делают одно и то же, но по-разному. Функция `perror()` принимает строковый аргумент и не имеет возвращаемого значения:

```
#include <stdio.h>
void perror(const char *s);
```

Аргумент `perror()` — это строка, предоставленная пользователем. Когда вызывается `perror()`, он печатает эту строку, за которой следует двоеточие и пробел, а затем описание типа ошибки на основе значения `errno`. Вот простой пример использования `perror()`:

```
целочисленный основной(void)
{
    ФАЙЛ *fp;

    fp = fopen("/foo/bar", "r");

    if (fp == NULL)
        perror("Я нашел ошибку");

    возврат 0;
}
```

Листинг 7-3: `perror-example.c`

Если в вашей системе нет файла `/foo/bar`, вывод будет выглядеть следующим образом:

```
$ ./a.out Я
обнаружил ошибку: Нет такого файла или каталога
```

Вывод `perror()` идет в стандартную ошибку. Помните об этом, если вы хотите перенаправить вывод ошибок вашей программы в файл.

Еще одна функция, которая помогает преобразовывать коды ошибок в описательные сообщения, — это `strerror()`:

```
#include <string.h> char
*strerror(int errnum);
```

Эта функция принимает значение errno в качестве аргумента и возвращает строку который описывает ошибку. Вот пример использования strerror():

```
целочисленный основной(void)
{
    закрыть(5);
    printf("%s\n", strerror(errno)); вернуть 0;

}
```

Листинг 7-4: strerror-example.c

Вот результат работы этой программы:

```
$ ./a.out
Неверный дескриптор файла
```

7.4 Лучшая жизнь с strace и ltrace

Важно понимать разницу между библиотечными функциями и системными вызовами. Библиотечные функции являются более высокоуровневыми, полностью выполняются в пользовательском пространстве и предоставляют программисту более удобный интерфейс к функциям, которые выполняют реальную работу — системным вызовам. Системные вызовы работают в режиме ядра от имени пользователя и предоставляются ядром самой операционной системы. Библиотечная функция printf() может выглядеть как очень общая функция печати, но на самом деле она только форматирует данные, которые вы ей даете, в строки и записывает строковые данные с помощью низкоуровневого системного вызова write(), который затем отправляет данные в файл, связанный со стандартным выводом вашего терминала.

Утилита strace выводит каждый системный вызов, который делает ваша программа, вместе с его аргументами и возвращаемым значением. Хотите увидеть, какие системные вызовы делает printf()? Это просто! Напишите программу «Hello, world!», но запустите ее так:

```
$ strace ./a.out
```

Разве вас не впечатляет, насколько усердно работает ваш компьютер, чтобы просто напечатать? что-нибудь на экран?

Каждая строка вывода strace соответствует одному системному вызову. Большая часть вывода strace показывает вызовы mmap() и open() с именами файлов вроде ld.so и libc. Это связано с системными вещами, такими как отображение файлов на диске в память и загрузка общих библиотек. Скорее всего, вас все это не волнует. Для наших целей интерес представляют ровно две строки ближе к концу вывода:

```
write(1, "привет, мир\n", 12привет, мир) = 12
_exit(0) = ?
```

Эти строки иллюстрируют общий формат вывода strace :

- Имя вызываемой системной функции • Аргументы системного вызова в скобках • Возвращаемое значение системного вызова⁷
- после символа =

Вот и все, но какая кладезь информации! Вы также можете см. ошибки. В нашей системе, например, мы получаем следующие строки:

```
open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (Нет такого файла или каталога) open("/etc/ld.so.cache", O_RDONLY) = 3
```

Первый вызов open() пытается открыть файл с именем /etc/ld.so.preload . Возвращаемое значение вызова open() (которое должно быть неотрицательным дескриптором файла) равно -1, что указывает на какую-то ошибку. strace любезно сообщает нам, что ошибка, которая привела к сбою open(), — ENOENT: файл /etc/ld.so.preload не существует.

strace сообщает нам, что второй вызов open() вернул значение 3. Это допустимый дескриптор файла, поэтому, по-видимому, вызов открытия файла /etc/ld.so.cache был выполнен успешно. Соответственно, на второй строке вывода strace нет кодов ошибок .

Кстати, не беспокойтесь о таких ошибках. То, что вы видите, связано с загрузкой динамической библиотеки и не является ошибкой как таковой. Файл ld.so.preload можно использовать для переопределения общих библиотек системы по умолчанию. Поскольку у меня нет желания возиться с такими вещами, файл просто отсутствует в моей системе. По мере накопления опыта работы со strace вы будете все лучше и лучше отфильтровывать этот тип «шума» и концентрироваться на тех частях вывода, которые вам действительно интересны. У strace есть

несколько опций, которые вам понадобятся в тот или иной момент, поэтому мы кратко опишем их здесь. Если вы посмотрели на полный вывод strace программы «Hello, world!», вы могли заметить, что strace может быть немного ... многословным. Гораздо удобнее сохранить весь этот вывод в файле, чем пытаться просмотреть его на экране. Один из способов, конечно, перенаправить stderr, но вы также можете использовать переключатель -o logfile , чтобы заставить strace записывать весь свой вывод в файл журнала. Кроме того, strace обычно обрезает строки до 32 символов. Иногда это может скрыть важную информацию. Чтобы заставить strace обрезать строки до N символов, вы можете использовать параметр -s N . Наконец, если вы запускаете strace в программе, которая разветвляет дочерние процессы, вы можете записать вывод strace для отдельных дочерних процессов в файл с именем LOG.xxx с помощью ключа -o LOG -ff , где xxx — идентификатор дочернего процесса.

⁷ Вас может удивить вопросительный знак возвращаемого значения exit(). Все , что говорит strace здесь, это то, что _exit возвращает void.

Также есть утилита ltrace , которая похожа на strace, но показывает библиотеку вызовы, а не системные вызовы. У ltrace и strace много общих опций, поэтому знание того, как использовать один из них, значительно продвинет вас к изучению другого.

Утилиты strace и ltrace очень полезны, когда вы хотите отправить сообщение об ошибке. отчеты и диагностическую информацию для специалистов по сопровождению программ, исходного кода которых у вас нет, и даже если у вас есть исходные файлы, использование этих инструментов иногда может оказаться быстрее, чем изучение кода.

Один из авторов впервые наткнулся на полезность этих инструментов при попытке установить и запустить плохо документированное проприетарное приложение на своей системе. При запуске приложение немедленно возвращалось в оболочку, по-видимому, ничего не делая. Он хотел отправить компании что-то более информативное, чем просто замечание: «Ваша программа немедленно завершается». Запуск strace на приложении дал подсказку:

```
open(umovestr: Ошибка ввода/вывода 0, O_RDONLY) = -1 EFAULT (Неверный адрес)
```

и запуск ltrace дал еще больше подсказок:

fopen(NULL, "r")	= 0
------------------	-----

Согласно выходным данным, это был самый первый вызов fopen(). Предположительно, приложение хотело открыть какой-то файл конфигурации, но fopen() был передан NULL. У приложения был какой-то внутренний обработчик ошибок, который завершился, но не выдал сообщение об ошибке. Автор смог написать подробный отчет об ошибке в компанию, и, как оказалось, проблема заключалась в том, что приложение поставлялось с неисправным глобальным файлом конфигурации, который указывал на несуществующий локальный файл конфигурации. На следующий день был выпущен патч.

С тех пор авторы пришли к выводу, что strace и ltrace чрезвычайно полезны для отслеживания ошибок и выяснения упрямого, загадочного поведения программ, которое может вызывать массу недоумений.

7.5 Статические средства проверки кода: lint и его друзья

Существует ряд бесплатных и коммерческих инструментов, которые сканируют ваш код без его компиляции и предупреждают вас об ошибках, возможных ошибках и отклонениях от строгих стандартов кодирования C. Они называются статическими проверщиками кода. Канонический статический проверщик кода для C, написанный SC Johnson в конце 1970-х, назывался lint. Он был написан в основном для проверки вызовов функций, поскольку ранние версии C не поддерживали прототипирование. lint породил множество производных статических проверщиков. Один из таких проверщиков, написанный Dave Evans из факультета компьютерных наук Университета Вирджинии, назывался lclint и был популярен в современных системах, таких как Linux. В январе 2002 года Dave переименовал lclint в splint , чтобы подчеркнуть повышенное внимание к безопасному программированию (и потому что splint легче произносить, чем lclint).

Цель шины — помочь вам написать наиболее защитный, безопасный и возможна программа без ошибок. `splint`, как и его предшественники, может быть очень придирчивым о том, что представляет собой хороший код.⁸ В качестве упражнения попробуйте найти что-нибудь в следующий код, который может вызвать предупреждение:

```
int main(пустота)
{
    целочисленный i;

    scanf("%d", &i);

    возврат 0;
}
```

Листинг 7-5: `scan.c`

Когда этот код запускается через `splint`, он предупреждает, что вы отбрасываете возвращаемое значение `scanf()`.⁹

\$ тест шины.c

`Splint 3.0.1.6 --- 11 февраля 2002 г.`

`test.c: (функции main)`

`test.c:8:2: Возвращаемое значение (типа int) игнорируется: scanf("%d", &i)`

Результат, возвращаемый вызовом функции, не используется. Если это предусмотрено, можно привести результат (`void`) для устранения сообщения. (Используйте `-retvalint` для отключения предупреждения)

Проверка завершена --- 1 код предупреждения

Все предупреждения о шинах имеют фиксированный формат. Первая строка предупреждения о шинах сообщает вам имя файла и функцию, где возникает предупреждение. Следующая строка дает номер строки и позицию предупреждения. После этого следует описание предупреждения, а также инструкции о том, как подавить этот вид предупреждения. Как вы можете видеть здесь, вызов `splint -retvalint`

`test.c` отключает все предупреждения об отбрасывании возвращаемых значений целочисленных функций. В качестве альтернативы, если вы не хотите отключать все отчеты об отброшенных возвращаемые значения `int`, вы можете отключить предупреждение только для этого вызова `scanf()` путем приведения типа `scanf()` к `void`. То есть, замените `scanf("%d", &i);` на `(void) scanf("%d", &i);`. (Есть еще один способ подавить это предупреждение, используя аннотации, о которых заинтересованный читатель может узнать из шины (документация).)

⁸ Многие программисты считают почетным, когда `splint` не сообщает об отсутствии предупреждений. Когда это происходит, код объявляется безлинтным.

⁹ Возвращаемое значение `scanf()` — это количество назначенных входных элементов.

7.5.1 Как К

Используйте шину

В комплект шины входит ошеломляющее количество переключателей, но по мере ее использования вы почувствуйте, какие из них обычно соответствуют вашим потребностям. Многие переключатели являются логическими в природе — они включают или выключают функцию. Эти типы переключателей имеют предварительный знак + для включения или - для выключения. Например, -retvalint отключает отчетность об отброшенных возвращаемых значениях int , а +retvalint отключает эту функцию отчетность (что является поведением по умолчанию).

Существует два способа использования шины. Первый — неформальный и используется в основном когда вы запускаете шину на чужом коде или на коде, который в основном финал:

\$ шина +слабая *.c

Без переключателя +weak шина обычно слишком разборчива, чтобы быть полезной. три уровня проверки в дополнение к +weak. Вы можете получить больше информации о них можно узнать из руководства по шиномонтажу , но вкратце они следующие:

- +weak Слабая проверка, обычно для неаннотированного кода C
- +стандартный режим по умолчанию
- +проверки Умеренно строгая проверка
- +strict Нелепо строгая проверка

Более формальное использование splint подразумевает документирование вашего кода специально для использования с splint. Эта документация, специфичная для splint, называется an -notation. Аннотация — это большая тема, и она увела бы нас слишком далеко, чтобы включите его в эту книгу, но он хорошо описан в документации по шине .

7.5.2 Последний Слова

splint поддерживает многие, но не все, расширения библиотеки C99. Он также не поддерживает некоторые изменения языка C99. Например, splint не знает о сложных типах данных или об определении переменной int в инициализаторе цикла for .

splint выпускается под лицензией GNU GPL. Его домашняя страница находится по адресу <http://www.splint.org/>.

7.6 Отладка динамически выделяемой памяти

Как вы, возможно, знаете, динамически выделяемая память (DAM) — это память, которую программа запрашивает из кучи с помощью таких функций, как malloc() и calloc().

10

Динамически выделяемая память обычно используется для таких структур данных, как двоичные данные. деревья и связанные списки, и это также работает за кулисами, когда вы создаете объект в объектно-ориентированном программировании. Даже стандартная библиотека C

¹⁰

В оставшейся части этого раздела мы будем ссылаться только на malloc(), но на самом деле мы имеем в виду malloc() и друзья, такие как calloc() и realloc().

использует DAM для своих внутренних целей. Вы также можете вспомнить, что динамическая память должна быть освобождена, когда вы с ней закончили.¹¹ Проблемы DAM, как известно, трудно обнаружить, и они попадают в несколько общих категорий:

- Динамически выделенная память не освобождается.

Вызов malloc() завершается неудачей (это легко обнаружить, проверив возвращаемое значение malloc()).

- Чтение или запись выполняется по адресу за пределами сегмента DAM.
- Чтение или запись выполняется в память внутри области DAM после сегмент освобожден.
- free()

вызывается дважды для одного и того же сегмента динамической памяти.

Эти ошибки могут не привести к очевидному сбою вашей программы. Давайте обсудим это немного подробнее. Чтобы сделать обсуждение более конкретным, вот пример, иллюстрирующий эти проблемы:

```
int main(void) {

    int *a = (int *) malloc( 3*sizeof(int) ); // возврат malloc не проверяется int *b = (int
    * ) malloc( 3*sizeof(int) ); // возврат malloc не проверяется

    для (int i = -1; i <= 3; ++i)
        a[i] = i; // плохая запись для i = -1 и 3

    free(a);
    printf("%d\n", a[1]); // чтение из освобожденной памяти
    free(a); // двойное освобождение указателя a

    return 0; // программа завершается без освобождения *b.
}
```

Листинг 7-6: memprobs.c

Первая проблема называется утечкой памяти. Например, рассмотрим следующий код:

```
int main(void) {

    ... много предыдущего кода ...

    мояФункция();
```

¹¹ Одним из заметных исключений является функция alloca(), которая запрашивает динамическую память из текущего стекового фрейма, а не из кучи. Память в фрейме автоматически освобождается, когда функция возвращается. Таким образом, вам не нужно освобождать память, выделенную alloca().

```
... много будущего кода ...
}
```

```
void мояФункция(void)
{
    символ *имя = (символ *) malloc( 10 * размер(символ) );
}
```

Листинг 7-7: Пример утечки память

При выполнении myFunction() вы выделяете память для 10 значений char . Единственный способ обратиться к этой памяти — использовать адрес, возвращаемый функцией malloc(), который вы сохранили в переменной указателя name. Если по какой-то причине вы потеряете адрес — например, name выйдет из области видимости, когда myFunction() завершается, и вы не сохранили копию ее значения где-либо еще — тогда у вас нет возможности получить доступ к выделенной памяти, и, в частности, у вас есть нет возможности его освободить.

Но это именно то, что происходит в коде. Динамически выделенная память не исчезает просто так или выходит из области видимости, как хранилище для переменная, размещенная в стеке, как и name , и поэтому каждый раз, когда вызывается myFunction() , он поглощает память на 10 символов, которые затем никогда не освобождаются. В результате доступное пространство кучи становится все меньше и меньше. Вот почему это Ошибка называется утечкой памяти.

Утечки памяти уменьшают объем памяти, доступной программе. В большинстве современных систем, таких как GNU/Linux, эта память освобождается операционная система при завершении работы приложения с утечкой памяти. На древних системах, таких как Microsoft DOS и Microsoft Windows 3.1, произошла утечка Память теряется до перезагрузки операционной системы. В любом случае утечки памяти вызывают снижение производительности системы из-за увеличения подкачки. Со временем они могут привести к сбою программы, в которой произошла утечка, или даже всей системы.

Вторая проблема, возникающая при динамически выделяемой памяти заключается в том, что вызов malloc() может завершиться неудачей. Существует множество способов, которыми это может произойти. Например, ошибка в вычислениях может привести к запросу суммы DAM, который слишком большой или отрицательный. Или, возможно, система действительно вышла из строя памяти. Если вы не понимаете, что это произошло, и продолжаете пытаться читать и писать в то, что вы ошибочно считаете действительным DAM, осложнения развиваются в уже неприятной ситуации. Это тип доступа нарушение, которое мы обсудим в ближайшее время. Однако, чтобы избежать этого, вы всегда должны проверьте, возвращает ли malloc() указатель , отличный от NULL , и попробуйте выйти из если это не так, запрограммируйте изящно.

Третья и четвертая проблемы называются ошибками доступа. Обе по сути являются версиями одного и того же: программа пытается прочитать или записать на адрес памяти, который ему недоступен. Третья проблема заключается в доступ к адресу памяти выше или ниже сегмента DAM. Четвертый

проблема заключается в доступе к адресу памяти, который раньше был доступен, но был освобожден до попытки доступа.

Вызов free() для одного и того же сегмента DAM дважды в разговорной речи называется двойным освобождением. Библиотека C имеет внутренние структуры управления памятью, которые описывают границы каждого выделенного сегмента DAM. Когда вы вызываете free() дважды для одного и того же указателя на динамическую память, структура управления памятью программы портится, что может привести к сбою программы или, в некоторых случаях, может позволить злонамеренному программисту использовать ошибку для переполнения буфера. В некотором смысле, это также является нарушением прав доступа, но для самой библиотеки C, а не для программы.

Нарушения прав доступа могут привести к одному из двух событий: программа может сбить, возможная запись основного файла¹² (обычно после получения сигнала об ошибке сегментации), или, что еще хуже, он может продолжить выполнение, что приведет к повреждению данных.

Из этих двух последствий первое бесконечно более желательно. Фактически, существует ряд инструментов, которые заставят вашу программу сегментировать ошибку и делать дамп ядра всякий раз, когда обнаруживается какая-либо проблема с DAM, вместо того, чтобы рисковать альтернативой!

Вы можете задаться вопросом: «Зачем мне, черт возьми, хотеть, чтобы моя программа выдавала ошибку?» Вам следует свыкнуться с мыслью, что если обработка DAM вашим кодом глючит, то это Very Good Thing™, когда он дает сбой, потому что другой вариант — это прерывистое, загадочное и невоспроизводимое плохое поведение. Повреждение памяти может оставаться незамеченным в течение долгого времени, прежде чем его последствия проявятся. Часто проблема проявляется в частях вашей программы, которые находятся довольно далеко от ошибки, и ее отслеживание может быть кошмаром. Если этого недостаточно, повреждение памяти может привести к нарушениям безопасности. Приложения, известные тем, что вызывают переполнение буфера и двойное освобождение, в некоторых случаях могут быть использованы злонамеренными взломщиками для запуска произвольного кода и являются причиной многих уязвимостей безопасности операционной системы.

С другой стороны, когда ваша программа сегментирует ошибки и делает дамп ядра, вы можете выполнить вскрытие основного файла и узнать точный исходный файл и номер строки кода, вызвавшего ошибку сегмента. И это, дорогой читатель, предпочтительнее, чем поиск ошибок.

Короче говоря, крайне важно как можно скорее выявить проблемы с DAM.

7.6.1 Стратегии для обнаружения проблемы

В этом разделе мы обсудим Electric Fence, библиотеку, которая устанавливает «забор» вокруг выделенных адресов памяти. Доступ к памяти за пределами этих заборов обычно приводит к ошибке сегментации и дампу ядра. Мы также обсудим два инструмента GNU, mtrace() и MALLOC_CHECK_, которые добавляют хуки в стандартные функции выделения libc для хранения записей о текущей выделенной памяти. Это позволяет libc выполнять проверки памяти, которую вы собираетесь прочитать, записать или освободить.

Помните, что при использовании нескольких программных средств необходима осторожность, каждое из которых

¹² В разговорной речи это называется сбросом керна.

из которых использует хуки для вызовов функций, связанных с кучей, поскольку одно средство может установить один из своих хуков поверх ранее установленного хука.¹³

7.6.2 Электрический Игородъ

Electric Fence, или EFence, — это библиотека, написанная Брюсом Перенсом в 1988 году и выпущенная под лицензией GNU GPL, когда он работал в Pixar. При подключении к вашему коду она заставляет программу немедленно сегментировать неисправность и выдавать дамп core¹⁴, если происходит что-либо из следующего:

- Чтение или запись выполняются за пределами границ DAM.
- Чтение или запись выполняются в DAM, который уже освобожден.
- Функция free() выполняется для указателя, который не указывает на DAM, выделенный функцией malloc() (сюда входит двойное освобождение как особый случай).

Давайте посмотрим, как использовать Electric Fence для отслеживания проблем malloc(). Рассмотрим программу outOfBound.c:

```
целочисленный основной(void)
{
    int *a = (int *) malloc( 2*sizeof(int) );

    для (int i=0; i<=2; ++i) { a[i] =
        i;
        printf("%d\n ", a[i]);
    }

    свободно(a);
    вернуть 0;
}
```

Листинг 7-8: outOfBound.c

Хотя программа содержит архетипическую ошибку malloc(), она, вероятно, скомпилируется без предупреждений. Скорее всего, она даже запустится без проблем:¹⁵

```
$ gcc -g3 -Wall -std=c99 outOfBound.c -o outOfBound_without_efence -lefence $ ./outOfBound_without_efence
```

¹³ На самом деле, вы можете безопасно использовать mtrace() и MALLOC_CHECK_ вместе, поскольку mtrace() тщательно сохраняет все существующие хуки, которые он находит.

¹⁴ Если вы запускаете программу, связанную с EFence, из GDB, а не вызываете ее в командной строке, программа выполнит seg fault без дампа ядра. Это желательно, поскольку файлы core исполняемых файлов, связанных с EFence, могут быть довольно большими, и вам в любом случае не нужен файл core, поскольку вы уже будете внутри GDB и будете смотреть на файл исходного кода и номер строки, где произошел seg fault.

¹⁵ Это не значит, что переполнение malloc() не нанесет ущерб вашему коду! Этот пример придуман, чтобы показать, как использовать EFence. В реальной программе запись за пределами массива может вызвать серьезные проблемы!

0
1
2

Мы смогли записать данные за пределы последнего элемента массива `a[]`. Сейчас все выглядит нормально, но это просто означает, что эта ошибка проявляется непредсказуемо и ее будет трудно обнаружить позже.

Теперь мы свяжем `outOfBound` с `EFence` и запустим его. По умолчанию `EFence` перехватывает только чтение или запись за пределами последнего элемента динамически выделенного региона. Это означает, что `outOfBound` должен выдать ошибку сегмента при попытке записи в `a[2]`:

```
$ gcc -g3 -Wall -std=c99 outOfBound.c -o outOfBound_with_efence -lefence $ ./outOfBound_with_efence
```

Электрический забор 2.1 Авторские права (C) 1987-1998 Брюс Перенс.

0
1

Ошибка сегментации (сброс керна)

Конечно же, `EFence` обнаружил операцию записи за пределами последнего элемента массива.

Случайный доступ к памяти до первого элемента массива (например, указание «элемента» `a[-1]`) встречается реже, но это, безусловно, может произойти в результате ошибочных вычислений индекса. `EFence` предоставляет глобальный `int` с именем `EF_PROTECT_BELOW`. Если установить эту переменную в 1, `EFence` будет отлавливать только недозаполнение массива и не будет проверять переполнение массива:

```
extern int EF_PROTECT_BELOW;
```

```
двойная мояФункция(void) {
    EF_PROTECT_BELOW = 1; // Проверка снизу
    int *a = (int *) malloc( 2*sizeof(int) );
    для (int i=-2; i<2; ++i) { a[i] = i;
        printf("%d\n", a[i]);
    }
    ...
}
```

Благодаря особенностям работы `EFence` вы можете перехватывать как попытки доступа к памяти за пределами динамически выделенных блоков, так и попытки доступа к памяти до выделенных блоков, но не оба типа ошибок доступа одновременно.

Для большей точности вам следует дважды запустить вашу программу с помощью EFence: один раз в режиме по умолчанию для проверки переполнения динамической памяти и второй раз EF_PROTECT_BELOW, установленным в 1, для проверки недозагрузки.¹⁶

Помимо EF_PROTECT_BELOW, EFence имеет несколько других глобальных целочисленных переменных, которые можно задать для управления его поведением:

EF_DISABLE_BANNER Установка этой переменной в 1 скрывает показываемый баннер.

воспроизводится при запуске программы, связанной с EFence. Делать это не рекомендуется, поскольку баннер предупреждает вас о том, что EFence связан с приложением и что исполняемый файл не должен использоваться для выпуска в производство, поскольку исполняемые файлы, связанные с EFence, больше по размеру, работают медленнее и создают очень большие файлы ядра.

EF_PROTECT_BELOW Как уже обсуждалось, EFence по умолчанию проверяет наличие переполнения DAM.

Установка этой переменной в 1 заставит EFence проверять нехватку памяти. бежит.

EF_PROTECT_FREE По умолчанию EFence не будет проверять доступ к DAM, который уже был освобожден. Установка этой переменной в 1 включает защиту освобожденной памяти.

EF_FREE_WIPES По умолчанию EFence не будет изменять значения, хранящиеся в освобожденной памяти. Установка этой переменной в ненулевое значение заставляет EFence заполнять сегменты динамически выделенной памяти 0x0 перед их освобождением. Это упрощает обнаружение EFence несанкционированного доступа к освобожденной памяти.

EF_ALLOW_MALLOC_0 По умолчанию EFence будет перехватывать любой вызов malloc() с аргументом 0 (т. е. любой запрос на ноль байтов памяти).

Обоснование в том, что запись чего-то вроде char *p = (char *) malloc(0); вероятно, является ошибкой. Однако, если по какой-то причине вы действительно хотите передать ноль в malloc(), то установка этой переменной в ненулевое значение заставит EFence игнорировать такие вызовы.

В качестве упражнения попробуйте написать программу, которая обращается к уже освобожденному DAM, и используйте EFence для обнаружения ошибки.

Всякий раз, когда вы меняете одну из этих глобальных переменных, вам нужно перекомпилировать программу, что может быть неудобно. К счастью, есть более простой способ. Вы также можете задать переменные окружения оболочки с теми же именами, что и глобальные переменные EFence. EFence обнаружит переменные оболочки и выполнит соответствующее действие.

В качестве демонстрации мы установим переменную окружения EF_DISABLE_BANNER для подавления печати страницы баннера EFence. (Как упоминалось ранее, этого делать не следует; делайте, как я говорю, а не как я делаю!) Если вы используете Bash, выполните

\$ экспорт EF_DISABLE_BANNER=1

Пользователи оболочки C должны выполнить

¹⁶ Если вы хотите быть действительно осторожными, прочтите «Выравнивание слов и обнаружение выхода за пределы» и «Выход за пределы». разделы «Инструкции по отладке вашей программы» на странице руководства EFence.

```
% setenv EF_DISABLE_BANNER 1
```

Затем повторно запустите листинг 7-8 и убедитесь, что баннер отключен.
Другой трюк — установить переменные EFence из GDB во время сеанса отладки. Это работает, поскольку переменные EFence являются глобальными; однако, это также означает, что программа должна выполняться, но приостановлена.

7.6.3 Отладка Платформа Проблемы с ГНУ С Библиотека Инструменты

Если вы работаете на платформе GNU, например GNU/Linux, есть некоторые возможности библиотеки GNU C, похожие на EFence, которые можно использовать для перехвата и восстановиться после проблем с динамической памятью. Мы кратко обсудим их здесь.

7.6.3.1 Переменная среды MALLOC_CHECK_

Библиотека GNU C предоставляет переменную среды оболочки с именем MALLOC_CHECK_, который можно использовать, как EFence, для обнаружения нарушений доступа DAM, но вы не необходимо перекомпилировать вашу программу, чтобы использовать ее. Настройки и их эффекты следующее:

- 0 Все проверки DAM отключены (это также имеет место, если переменная неопределенная).
- 1 Диагностическое сообщение выводится на stderr при обнаружении повреждения кучи. проверено.
- 2 Программа немедленно прерывает работу и делает дамп ядра при повреждении кучи. обнаружена.
3. Совокупность эффектов 1 и 2.

17

Поскольку MALLOC_CHECK_ — это переменная среды, использовать ее для поиска проблем, связанных с кучей, так же просто, как ввести:

```
$ экспорт MALLOC_CHECK_=3
```

Хотя MALLOC_CHECK_ удобнее в использовании, чем EFence, у него есть несколько серьезных недостатков. Во-первых, MALLOC_CHECK_ сообщает о проблеме с динамической памятью только при следующем выполнении функции, связанной с кучей (например, (например , malloc() , realloc() или free()) после несанкционированного доступа к памяти. Это означает, что вы не только не знаете исходный файл и номер строки, проблемного кода, вы часто даже не знаете, какой указатель является проблемной переменной. Для иллюстрации рассмотрим этот код:

¹ целочисленный основной(пустой)

² {

¹⁷ Это не документировано в системе авторов. Спасибо Джанлуке Инсолвили за прочтение Исходники glibc и поиск этой опции!

```

3     int *p = (int *) malloc(sizeof(int));
4     int *q = (int *) malloc(sizeof(int));
5
6     для (целое i=0; i<400; ++i)
7         p[я] = я;
8
9     д[0] = 0;
10
11    свободно(q);
12    бесплатно(p);
13    возврат 0;
14 }

```

Листинг 7-9: malloc-check-0.c

Программа прерывается на строке 11, хотя проблема действительно возникает на строке 7. Изучение файла ядра может привести вас к выводу, что проблема кроется в q, не p:

```

$ MALLOC_CHECK_=3 ./malloc-check-0
malloc: использование отладочных хуков
free(): неверный указатель 0x8049680!
Прервано (сброс ядра)
$ gdb malloc-check-0 ядро
Ядро было сгенерировано `./malloc-check-0'.
Программа завершена сигналом 6, прервано.
Чтение символов из /lib/libc.so.6...выполнено.
Загруженные символы для /lib/libc.so.6
Чтение символов из /lib/ld-linux.so.2...выполнено.
Загруженные символы для /lib/ld-linux.so.2
#0 0x40046a51 в kill() из /lib/libc.so.6
(гдб) бт
#0 0x40046a51 в kill() из /lib/libc.so.6
#1 0x40046872 в raise() из /lib/libc.so.6
#2 0x40047986 в abort () из /lib/libc.so.6
#3 0x400881d2 в _IO_file_xsputn () из /lib/libc.so.6
#4 0x40089278 в свободном () из /lib/libc.so.6
#5 0x080484bc в main () в malloc-check-0.c:13

```

С этим недостатком можно смириться при отладке 14-строчного кода. программы, но это может быть серьезной проблемой при работе с реальным приложением. Тем не менее, знание того, что проблема DAM вообще существует, полезно информация.

Во-вторых, это подразумевает, что если после вызова не вызывается функция, связанная с кучей, если произойдет ошибка доступа, MALLOC_CHECK_ вообще не сообщит об ошибке.

В-третьих, сообщения об ошибках MALLOC_CHECK_ не кажутся очень осмысленными. Хотя в предыдущем листинге программы была ошибка переполнения массива, Сообщение об ошибке было просто «неверный указатель». Технически верно, но бесполезно.

Наконец, `MALLOC_CHECK_` отключен для программ `setuid` и `setgid`, поскольку эта комбинация функций может быть использована в эксплойте безопасности. Его можно включить повторно, создав файл `/etc/suid-debug`. Содержимое этого файла не важно, важно только существование файла.

В заключение, `MALLOC_CHECK_` — удобный инструмент для использования во время разработки кода, чтобы обнаружить ошибки программирования, связанные с кучей. Однако, если вы подозреваете проблему DAM или хотите тщательно просканировать свой код на предмет возможных проблем DAM, вам следует использовать другую утилиту.

7.6.3.2 Использование `mcheck()` Facility

Альтернативой `MALLOC_CHECK_` для обнаружения проблем DAM является `mcheck()`. Мы обнаружили, что этот метод более удовлетворительный, чем `MALLOC_CHECK_`.

Прототипом `mcheck()` является

```
#include <mcheck.h>
int mcheck (void (*ABORTHANDLER) (enum mcheck_status CTATUS))
```

Вы должны вызвать `mcheck()` перед вызовом любых функций, связанных с кучей, в противном случае вызов `mcheck()` завершится ошибкой. Поэтому эта функция должна быть вызвана в самом начале вашей программы. Вызов `mcheck()` возвращает 0 в случае успеха и -1, если он вызван слишком поздно.

Аргумент `*ABORTHANDLER` — это указатель на пользовательскую функцию, которая вызывается при обнаружении несоответствия в DAM. Если вы передаете NULL в `mcheck()`, то используется обработчик по умолчанию. Как и `MALLOC_CHECK_`, этот обработчик по умолчанию выводит сообщение об ошибке на `stdout` и вызывает `abort()` для создания файла ядра. В отличие от `MALLOC_CHECK_`, сообщение об ошибке полезно. Например, выход за пределы динамически выделенного сегмента в следующем примере:

```
целочисленный основной(void)
{
    mcheck(NULL);
    int *p = (int *) malloc(sizeof(int)); p[1] = 0;
    free(p);
    return 0;
}
```

Листинг 7-10: `mcheckTest.c`

выдает сообщение об ошибке, показанное здесь:

```
$ gcc -g3 -Wall -std=c99 mcheckTest.c -o mcheckTest -lmcheck $ ./mcheckTest
память затерта
за пределами выделенного блока Прервано (сделан
дамп ядра)
```

Другие типы проблем сопровождаются аналогичными описательными сообщениями об ошибках.

7.6.3.3 Использование mtrace() для обнаружения утечек памяти и двойного освобождения памяти

Функция mtrace() является частью библиотеки GNU C и используется для обнаружения утечек памяти и двойного освобождения в программах C и C++. Ее использование включает пять шагов:

1. Установите переменную окружения MALLOC_TRACE на допустимое имя файла. Это имя файла, в который mtrace() помещает свои сообщения. Если эта переменная не задана допустимое имя файла или для файла не установлены разрешения на запись, mtrace() ничего не делает.
2. Включите заголовочный файл mcheck.h .
3. Вызовите mtrace() в верхней части вашей программы. Его прототип —

```
#include <mcheck.h>
недействительный mtrace(void);
```

4. Запустите программу. Если будут обнаружены какие-либо проблемы, они будут задокументированы, в нечитаемой человеком форме в файле, на который указывает MALLOC_TRACE. Также, по соображениям безопасности, mtrace() ничего не делает для setuid или setgid. исполняемые файлы.
5. Функция mtrace() поставляется с Perl-скриптом mtrace , который используется для анализирует файл журнала и выводит его содержимое на стандартный вывод в удобной для восприятия форме.

Обратите внимание, что есть также вызов muntrace() , который используется для остановки памяти. трассировка, но на странице информации glibc рекомендуется не использовать ее. Библиотека C, который также может использовать DAM для вашей программы, уведомляется о том, что ваша программа имеет завершается только после возврата из main() или вызова exit() .

Память, которую библиотека C использует для вашей программы, не освобождается до тех пор, пока не будет выполнено это действие. происходит. Вызов muntrace() до освобождения этой памяти может привести к ложному позитиву.

Давайте рассмотрим простой пример. Вот код, который иллюстрирует обе проблемы, которые mtrace() ловит. В следующем коде мы никогда не освобождаем память, выделенную в строке 6 и указанную p, а в строке 10 мы вызовите free() для указателя q, даже если он не указывает на динамически выделенную память.

¹ целочисленный основной(пустой)

```
2 {
3     целое *p, *q;
4
5     mtrace();
6     p = (int *) malloc(sizeof(int));
7     printf("p указывает на %p\n", p);
8     printf("q указывает на %p\n", q);
9
10    свободно(q);
11    возврат 0;
```

12 }

Листинг 7-11: mtrace1.c

Мы компилируем эту программу и запускаем ее, предварительно установив переменную `MALLOC_TRACE`.

```
$ gcc -g3 -Wall -Wextra -std=c99 -o mtrace1 mtrace1.c
$ MALLOC_TRACE="./mtrace.log" ./mtrace1
р указывает на 0x8049a58
q указывает на 0x804968c
```

Если посмотреть на содержимое `mtrace.log`, то оно вообще не имеет смысла. Однако запуск скрипта Perl `mtrace()` дает понятный вывод:

```
$ cat mtrace.log
= Начать
@ ./mtrace1:(mtrace+0x120)[0x80484d4] + 0x8049a58 0x4
@ ./mtrace1:(mtrace+0x157)[0x804850b] - 0x804968c
p@satan$ mtrace mtrace.log
- 0x0804968c Свободно 3 никогда не было выделено 0x804850b
```

Память не освобождена:

Адрес	Размер	Звонящий
0x08049a58	0x4 в 0x80484d4	

Однако это лишь немного помогает, поскольку, хотя `mtrace()` и обнаружил проблемы, он сообщил о них как о адресах указателей. К счастью, `mtrace()` может сделать лучше. Скрипт `mtrace()` также принимает имя исполняемого файла как необязательный аргумент. Используя эту опцию, вы получаете номера строк вместе с сопутствующие проблемы.

```
- 0x0804968c Свободно 3 никогда не было выделено
/home/p/codeTests/mtrace1.c:15
```

Память не освобождена:

Адрес	Размер	Звонящий
0x08049a58	0x4 в /home/p/codeTests/mtrace1.c:11	

Вот это то, что мы хотели увидеть!

Как и утилиты `MALLOC_CHECK_` и `mcheck()`, `mtrace()` не помешает вашему программе от сбоя. Он просто проверяет наличие проблем. Если ваша программа сбои, часть выходных данных `mtrace()` может быть потеряна или искажена, что может выдавать загадочные отчеты об ошибках. Лучший способ справиться с этим — перехват и обработка ошибок сегмента для того, чтобы дать `mtrace()` шанс завершить работу изящно. Следующий пример иллюстрирует, как это сделать.

```
недействительный sigsegv_handler (INT Signum);
```

```
целочисленный основной(void)
{
    целое число *р;

    сигнал(SIGSEGV, sigsegv_handler); mtrace(); р =
    (int *)
    malloc(sizeof(int));

    поднять(SIGSEGV);
    вернуть 0;
}

void sigsegv_handler (int Signum) {

    printf("Перехвачен sigsegv: сигнал %d. Корректное завершение работы.\n", signum); muntrace(); abort();

}
```

Листинг 7-12: mtrace2.c

8

ИСПОЛЬЗОВАНИЕ GDB/DDD/ECLIPSE ДЛЯ ДРУГИЕ ЯЗЫКИ



GDB и DDD широко известны как отладчики для программ на C/C++, но они может быть использован для разработки на других языках. Eclipse изначально был разработан для разработки на Java, но имеет плагины для многих других языки. Эта глава покажет вам, как использовать этот многоязычность.

GDB/DDD/Eclipse не обязательно являются «лучшими» отладчиками для любого конкретного языка. Доступно большое количество отличных инструментов отладки для определенных языков. Но мы говорим, что было бы неплохо иметь возможность использовать один и тот же интерфейс отладки независимо от того, на каком языке вы работаете писать на C, C++, Java, Python, Perl или других языках/отладчиках, с которыми могут использоваться эти инструменты. DDD был «перенесен» на все их.

Например, рассмотрим Python. Интерпретатор Python включает в себя простой текстовый отладчик. Опять же, существует ряд отличных отладчиков и IDE, специфичных для Python, но другой вариант — использовать DDD как интерфейс для встроенного отладчика Python. Это позволяет вам достичь

удобство графического пользовательского интерфейса при использовании интерфейса, знакомого вам по программированию на C/C++ (DDD).

Каким образом достигается многоязыковая универсальность этих инструментов?

- Хотя GDB изначально был создан как отладчик для C/C++, позже разработчики GNU предложили также компилятор Java, GCJ.
- Помните, что DDD — это не отладчик сам по себе, а скорее графический интерфейс пользователя. через который вы можете давать команды базовому отладчику. Для С и С++ таким базовым отладчиком обычно является GDB. Однако DDD может использоваться и часто используется в качестве интерфейса для отладчиков, специфичных для других языков.
- Eclipse также является просто интерфейсом. Плагины для различных языков дают ему возможность управлять разработкой и отладкой кода на этих языках.

В этой главе мы дадим обзор отладки в Java, Perl, Python и языке ассемблера с помощью этих инструментов. Следует отметить, что в каждом случае есть дополнительные функции, не рассмотренные здесь, и мы настоятельно рекомендуем вам изучить детали для используемого вами языка.

8.1 Ява

В качестве примера рассмотрим прикладную программу, которая манипулирует связанным списком. Здесь объекты класса Node представляют узлы в связанном списке чисел, которые поддерживаются в порядке возрастания значения ключа. Сам список является объектом класса LinkedList. Тестовая программа TestLL.java считывает числа из командной строки, создает связанный список, состоящий из этих чисел в отсортированном порядке, а затем выводит отсортированный список. Вот исходные файлы:

TestLL.java

```

1 // использование: [java] TestLL список_тестовых_целых_значений
2
3 // простой пример программы; считывает целые числа из командной строки, 4 //
сохраняет их в линейном связанным списке, поддерживая возрастающий порядок, 5 // а затем
6 // выводит окончательный список на экран
7
8 public class TestLL {
9     public static void main(String[] Args) { int NumElements
10         = Args.length; LinkedList LL = new
11         LinkedList(); for (int I = 1; I <= NumElements; I+
12         ) { int Num; // выполняем "atoi()" языка С, используя
13         parseInt()
14
15         Num = Integer.parseInt(Args[I-1]); Узел NN =
16         новый Узел (Num);

```

```

17         LL.Вставить(NN);
18     }
19     System.out.println("окончательный отсортированный список:");
20     LL.PrintList();
21 }
22 }
```

LinkedList.java

```

1 // LinkedList.java, реализующий упорядоченный связанный список целых чисел
2
3 публичный класс LinkedList
4 {
5     публичный статический заголовок узла = null;
6
7     публичный СвязанныйСписок() {
8         Голова = Ноль;
9     }
10
11    // вставляет узел N в этот список
12    public void Вставить(Узел N) {
13        если (Голова == null) {
14            Голова = N;
15            возвращаться;
16        }
17        если (N.Значение < Head.Значение) {
18            N.Next = Голова;
19            Голова = N;
20            возвращаться;
21        }
22        иначе если (Head.Next == null) {
23            Head.Next = N;
24            возвращаться;
25        }
26        для (Узел D = Голова; D.Следующий != null; D = D.Следующий) {
27            если (N.Значение < D.Следующее.Значение) {
28                N.Следующий = D.Следующий;
29                D.Следующий = N;
30                возвращаться;
31            }
32        }
33    }
34
35    публичный статический void PrintList() {
36        если (Head == null) возврат;
37        для (Узел D = Голова; D != null; D = D.Следующий)
38            System.out.println(D.Value);
```

```

39      }
40  }

```

узел.java

```

1 // Node.java, класс для узла в упорядоченном связанным списке целых чисел
2
3 публичный класс Node
4  {
5      Значение int;
6      Узел Next; // "указатель" на следующий элемент в списке
7
8      // конструктор
9      публичный узел (int V) {
10         Значение = V;
11         Далее = ноль;
12     }
13 }

```

В коде есть ошибка. Давайте попробуем ее найти.

8.1.1 Прямой использование ГБД для Отладка Ява

Java обычно рассматривается как интерпретируемый язык, но с GNU GCJ компилятор, вы можете скомпилировать исходный код Java в машинный код. Это позволяет ваши Java-приложения будут работать намного быстрее, и это также означает, что вы можете использовать GDB для отладки. (Убедитесь, что у вас GDB версии 5.1 или более поздней.) GDB, напрямую или через DDD, является более мощным, чем JDB, отладчик, который поставляется с Java Development Kit. Например, JDB не позволяет вам для установки условных точек останова, что является базовой техникой отладки GDB, как вы видели. Таким образом, вы не только выигрываете, изучая на одного отладчика меньше, но и получаете лучшую функциональность.

Сначала скомпилируйте приложение в машинный код:

```
$ gcj -c -g Node.java
$ gcj -c -g СвязанныйСписок.java
$ gcj -g --main=TestLL TestLL.java Node.o LinkedList.o -o TestLL
```

Эти строки аналогичны обычным командам GCC, за исключением `-main=TestLL` опция, которая указывает класс, функция `main()` которого должна быть точка входа для выполнения программы. (Мы скомпилировали два исходных файла по одному за раз. Мы обнаружили, что это необходимо для того, чтобы гарантировать, что GDB корректно отслеживает исходные файлы.) Тестовый запуск программы ввод дает следующее:

```
$ ТестЛЛ 8 5 12
```

Окончательный отсортированный список:

5

8

Каким-то образом исчез ввод 12. Давайте посмотрим, как использовать GDB для поиска ошибки. Запустите GDB как обычно и сначала скажите ему не останавливаться и не выводить объявления на экран, когда сигналы Unix генерируются операциями по сбору мусора Java. Такие действия являются помехой и могут помешать вашей возможность пошагового использования GDB.

(gdb) обрабатывать SIGPWR nostop noprint			
Сигнал	останавливаться	Печать	Перейти к программе Описание
SIGPWR	Нет	Нет	Да
Сбой питания/перезапуск			
(gdb) обрабатывать SIGXCPU nostop noprint			
Сигнал	останавливаться	Печать	Перейти к программе Описание
SIGXCPU	Нет	Нет	Да
Превышен лимит времени ЦП			

Теперь, поскольку первой явной жертвой ошибки стало число 12 в входные данные, давайте установим точку останова в начале метода `Insert()`, при условии, что значение ключа узла равно 12:

```
(gdb) b LinkedList.java:13 если N.Value == 12
Точка останова 1 по адресу 0x8048bb4: файл LinkedList.java, строка 13.
```

В качестве альтернативы вы можете попробовать команду

```
(gdb) b LinkedList.java:Вставить, если N.Value == 12
```

Однако, хотя это сработает позже, на данном этапе `LinkedList` класс еще не загружен.

Теперь запустите программу в GDB:

```
(гдб) р 8 5 12
Запуск программы: /debug/TestLL 8 5 12
[Отладка потоков с использованием libthread_db включена]
[Новая тема -1208596160 (LWP 12846)]
[Новая тема -1210696800 (LWP 12876)]
[Переключение на тему -1208596160 (LWP 12846)]
```

Точка останова 1, `LinkedList.Вставить(Узел)` (`this=@47da8, N=@11a610`)
в `LinkedList.java:13`
13 если (`Голова == null`) {
Текущий язык: авто; в настоящее время java

Вспоминая принцип подтверждения, давайте подтвердим, что значение сейчас будет вставлено 12:

```
(gdb) р N.Значение
$1 = 12
```

Теперь давайте разберем код:

```
(гdb) н
17         если (N.Значение < Head.Значение) {
(гdb) н
22         иначе если (Head.Next == null) {
(гdb) н
26         для (Узел D = Голова; D.Следующий != null; D = D.Следующий) {
(гdb) н
27             если (N.Значение < D.Следующее.Значение) {
(gdb) p D.Следующее.Значение
2$ = 8
(гdb) н
26         для (Узел D = Голова; D.Следующий != null; D = D.Следующий) {
(гdb) н
12         public void Вставить(Узел N) {
(гdb) н
33     }
(гdb) н
TestLL.main(java.lang.String[])
Args=@ab480 в TestLL.java:12
12         for (int I = 1; I <= NumElements; I++) {
```

Хм, это нехорошо. Мы прошли через все Insert() без вставки
12.

Присмотревшись повнимательнее, вы увидите, что в цикле, начинающемся со строки 26, LinkedList.java мы сравнили значение , которое должно быть вставлено, 12, с двумя текущими значениями в списке, 5 и 8, и обнаружили, что в обоих случаях новое значение было больше. Вот в этом и заключается ошибка. Мы не занимались этим делом в котором значение, которое нужно вставить, больше, чем все значения, которые уже есть в list. Нам нужно добавить код после строки 31, чтобы обработать эту ситуацию:

```
иначе если (D.Следующий.Следующий == null) {
    D.Следующий.Следующий = N;
    возвращаться;
}
```

После внесения этого исправления вы обнаружите, что программа работает так, как и должна. должен.

8.1.2 ДДДс ГБД в Ява

С использованием

Отладывать

Эти шаги будут намного проще и приятнее, если вы используете DDD в качестве интерфейса к GDB (опять же, предполагая, что исходный код скомпилирован с использованием GCJ). Начало вверх DDD как обычно:

```
$ ddd TestLL
```

(Игнорируйте сообщение об ошибке временного файла.)

Исходный код не сразу появляется в исходном тексте DDD.
окно, поэтому вы можете начать с использования консоли DDD:

```
(gdb) обрабатывать SIGPWR nostop noprint
(gdb) обрабатывать SIGXCPU nostop noprint
(gdb) b СвязанныйСписок.java:13
Точка останова 1 по адресу 0x8048b80: файл LinkedList.java, строка 13.
(gdb) cond 1 N.Value == 12
(гдб) р 8 5 12
```

В этот момент появляется исходный код, и вы находитесь в точке останова.
Затем вы можете продолжить операции DDD в обычном режиме.

8.1.3 DDD как JDB

С использованием Графический интерфейс для

DDD можно использовать напрямую с отладчиком JDB из Java Development Kit.
Команда

```
$ ddd -jdb TestLL.java
```

запустит DDD, который затем вызовет JDB.

Однако мы обнаружили, что это становится довольно громоздким, поэтому мы
не рекомендуем такое использование DDD.

8.1.4 Отладка Java в Eclipse

Если вы изначально загрузили и установили версию Eclipse, предназначенную для разработки
на C/C++, вам потребуется получить JDT (Java Development
Инструменты) плагин.

Основные операции такие же, как мы описали ранее для C/C++, но обратите внимание на
следующий:

- При создании проекта обязательно выберите Java Project. Также мы
рекомендуем вам проверить Использовать папку проекта как корневую для источников и
Классы.
- Используйте представление Package Explorer для вашего навигатора.
- Исходные файлы будут скомпилированы в .class сразу после их сохранения (или
импортный).
- При создании диалогового окна запуска щелкните правой кнопкой мыши Java Application и выберите
Новый. В поле с надписью Main Class заполните класс, в котором находится main()
определенный.
- При создании диалогового окна отладки установите флагок Stop in Main.
- В отладочных запусках Eclipse остановится перед main(). Просто нажмите кнопку Resume-
тонна для продолжения.

Рисунок 8-1 показывает типичную сцену отладки Java в Eclipse. Обратите внимание, что как в C/C++ в представлении «Переменные» мы отобразили значения узла N направив треугольник рядом с буквой N вниз.

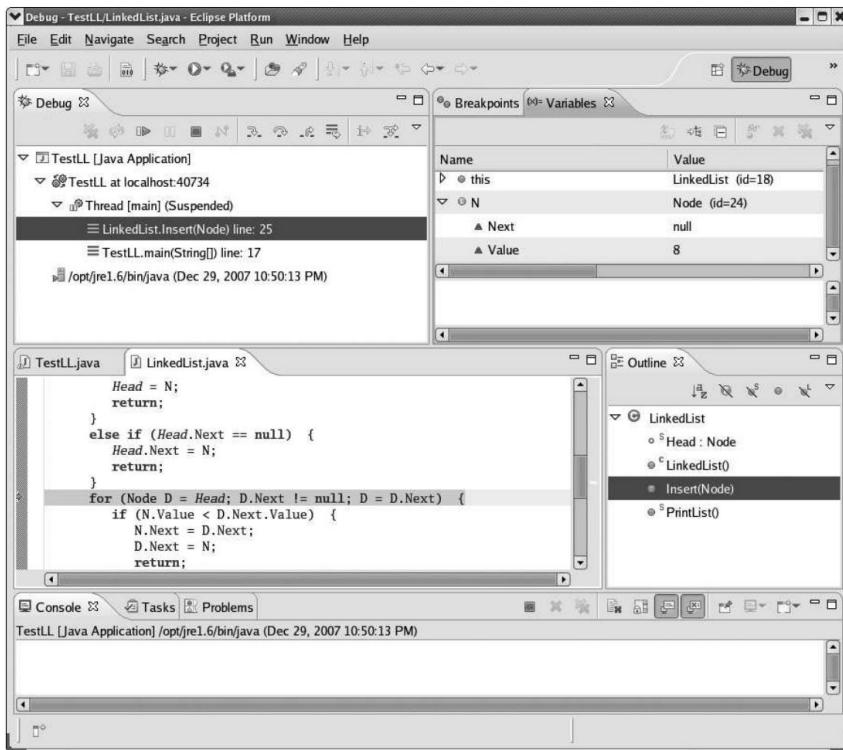


Рисунок8-1: Ява Вотладка Затмение

8.2 Перл

Мы будем использовать следующий пример, textcount.pl, который вычисляет статистику по текстовые файлы:

```

1 #!/usr/bin/perl
2
3 # читает текстовый файл, указанный в командной строке, и подсчитывает слова,
4 # строк и абзацев
5
6 $line_count = 0;
7
8 $word_count = 0;
9
10 $par_count = 0;
11

```

```

12 $now_in_par = 0; # сейчас не внутри абзаца
13
14 пока ($line = <INFILE>) {
15     $line_count++;
16     если ($line ne "\n") {
17         если ($now_in_par == 0) {
18             $par_count++;
19             $now_in_par = 1;
20         }
21         @words_on_this_line = split(" ",$line);
22         $word_count += скаляр(@words_on_this_line);
23     }
24     еще {
25         $now_in_par = 0;
26     }
27 }
28
29 print "$word_count $line_count $par_count\n";

```

Программа подсчитывает количество слов, строк и абзацев в тексте. текстовый файл, указанный в командной строке. В качестве тестового случая воспользуемся файлом test.txt показано ниже (вы можете узнать в нем текст из Главы 1):

В этой главе мы излагаем некоторые основные принципы отладки, как в целом, а также в отношении отладчиков GDB и DDD. По крайней мере один наших "правил" будут носить формальный характер, Основополагающий Принцип Отладка.

Начинающим, конечно, следует внимательно прочитать эту главу, так как Представленный здесь материал будет использоваться на протяжении всей оставшейся части книги.

Профессионалы могут поддаться соблазну пропустить главу. Мы предлагаем, однако, что они хотя бы бегло просматривают его. Многие профессионалы найдут в по крайней мере, какой-то новый материал, и в любом случае важно, чтобы все читатели начинают с общей предыстории.

Вверху одна пустая строка, две после Debugging и одна перед Профессионалы. Результатом запуска нашего кода Perl в этом файле должно быть как показано ниже:

```
$ perl textcount.pl test.txt
102 14 3
```

Теперь предположим, что мы забыли предложение else :

```

еще {
    $now_in_par = 0;
}
\end{Код}

```

Тогда выход будет таким:

```

\begin{Код}
$ perl textcount.pl test.txt
102 14 1

```

Количество слов и строк верное, но количество абзацев неверное.

8.2.1 Отладка Perl с помощью DDD

Perl имеет собственный встроенный отладчик, который вызывается с помощью опции -d в командной строке:

```
$ perl -d myprog.pl
```

Одним из недостатков встроенного отладчика является то, что у него нет графического интерфейса. но это можно исправить, запустив отладчик через DDD. Давайте посмотрим как мы могли бы использовать DDD для поиска ошибки. Вызовите DDD, набрав

```
$ ddd textcount.pl
```

DDD автоматически определяет, что это скрипт Perl, вызывает отладчик Perl и устанавливает зеленую стрелку «вы здесь» на первой исполняемой строке кода.

Теперь мы указываем аргумент командной строки, test.txt, нажав кнопку Program | Запуск и заполнение раздела «Запуск с аргументами» всплывающего окна окно, как показано на рисунке 8-2. (Вы можете получить сообщение в консоли DDD в духе «... не знаю, как создать новый TTY...» Просто (Игнорируйте его.) В качестве альтернативы мы могли бы задать аргумент «вручную», просто введя команду отладчика Perl

```
@ARGV[0] = "тест.txt"
```

в консоли DDD.

Поскольку ошибка в количестве абзацев, давайте посмотрим, что произойдет, когда программа дойдет до конца первого абзаца. Это произойдет прямо сейчас после состояния

```
$words_on_this_line[0] eq "Отладка."
```

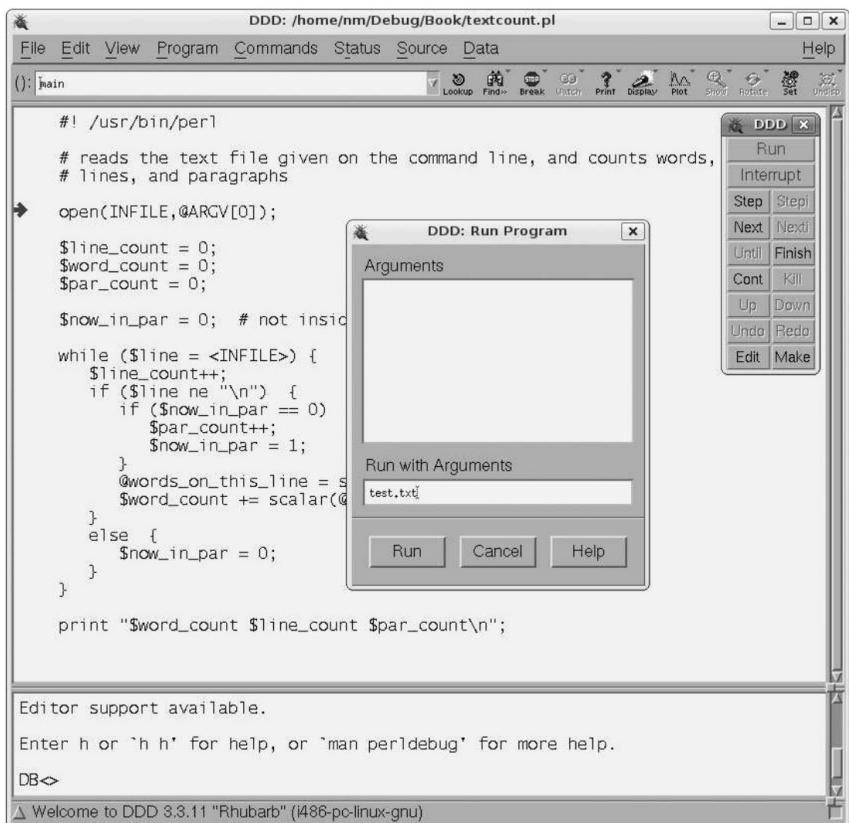


Рисунок 8-2: Установка аргументов командной строки

становится истинным. Давайте поставим точку останова около начала цикла while .

Мы делаем это точно так же, как мы это видели для программ на C/C++, щелкнув правой кнопкой мыши по строке и выбрав «Установить точку останова».

Мы также должны применить указанное выше условие к этой точке останова. Снова нажмите Source | Breakpoints, убедившись, что данная точка останова выделена во всплывающем окне Breakpoints and Watchpoints, а затем нажмите значок Props. Затем мы заполняем нужное условие. Смотрите рисунок 8-3.

Затем мы выбираем Program | Run. (Мы не выбираем Run Again, так как это, похоже, переносит нас во внутренние структуры Perl.) Мы перемещаем указатель мыши на экземпляр переменной \$words_on_this_line, и появляется обычное желтое поле DDD, отображающее значение этой переменной. Таким образом мы подтверждаем, что условие точки останова выполняется, как и должно быть. См. рисунок 8-4.

После нажатия кнопки «Далее» несколько раз, чтобы пропустить пустые строки в тексте файл, вы заметите, что мы также пропускаем строку

\$par_count++;

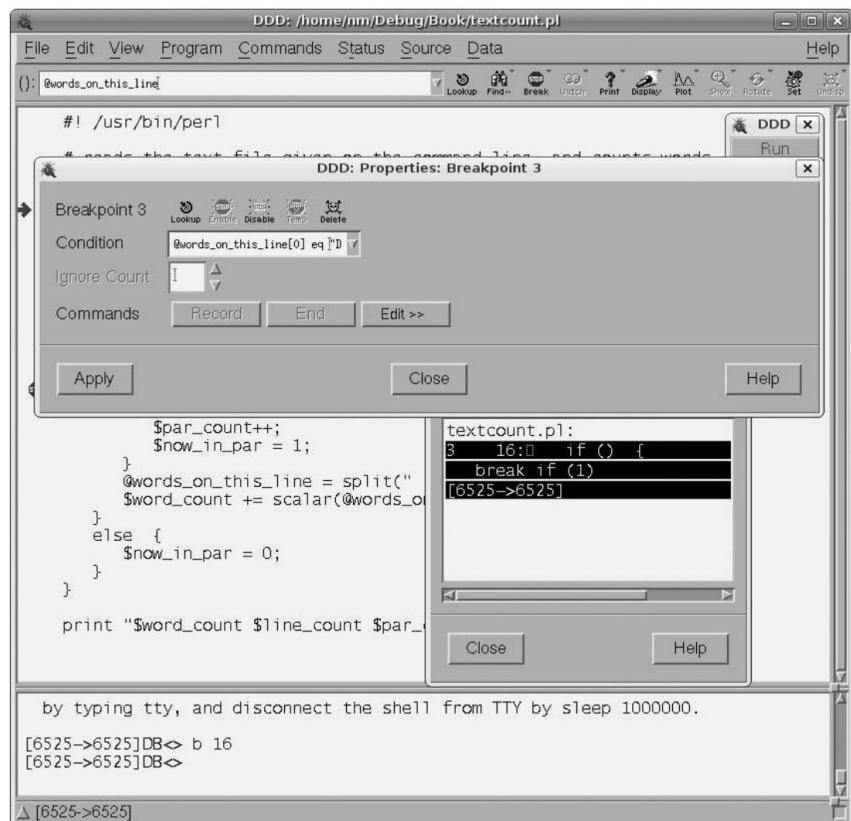


Рисунок8-3: Параметр а условия а на точка останова

который должен был увеличить количество абзацев. Двигаясь в обратном направлении, мы видим, что это было вызвано тем, что переменная \$now_in_par была равна 0, и, проанализировав это наблюдение, мы вскоре поймем, как исправить ошибку.

Точку останова также можно отключить, включить или удалить в DDD, выбрав Источник | Точки останова, выделив точку останова, а затем щелкнув желаемый выбор.

Если вы изменили исходный файл, вам необходимо уведомить DDD о необходимости обновления. выбираем Файл | Перезагрузить.

8.2.2 Отладка Perl в Eclipse

Для разработки кода Perl в Eclipse вам понадобится пакет PadWalker Perl, который вы можете загрузить с CPAN, и плагин EPIC Eclipse для Perl.

Опять же, основные операции те же самые, что мы описали ранее. C/C++, но обратите внимание на следующее:

- При создании проекта выберите Perl Project.

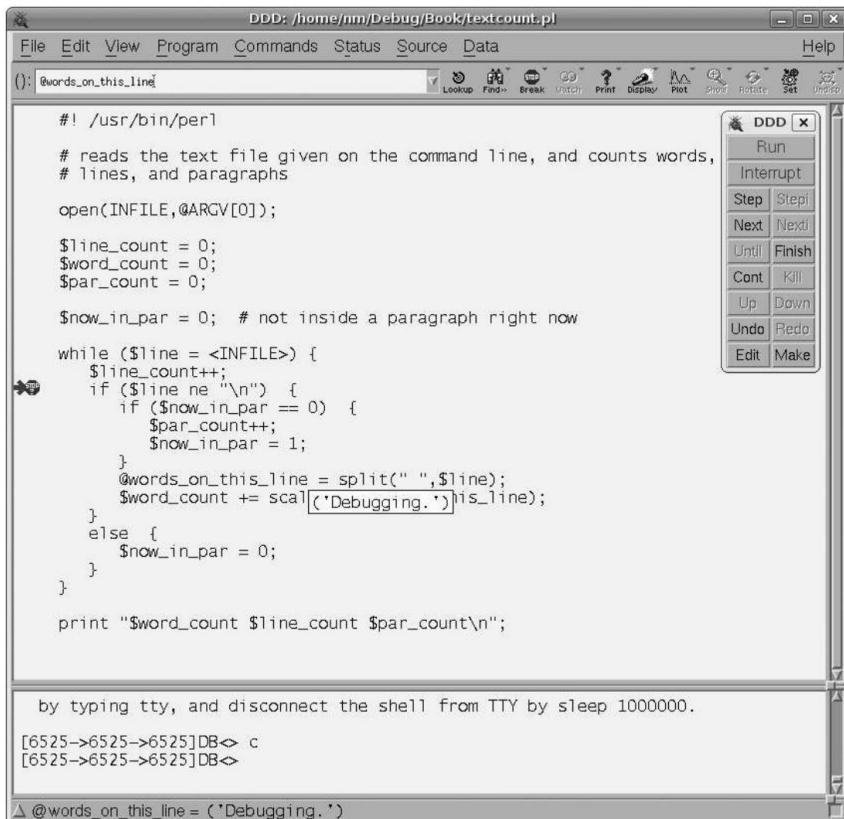


Рисунок8-4: По прибытии Вочки останова

- Используйте Навигатор в качестве вида навигатора.
- Фаза сборки отсутствует, поскольку Perl не создает байт-код.
- В перспективе отладки значения переменных доступны только в представлении «Переменные», а не с помощью мыши в представлении исходного кода, и только для локальных переменных.
Вам нужно будет использовать ключевое слово Perl my в первом экземпляре каждого переменную в исходном коде, чтобы увидеть глобальные переменные (это где есть пакет PadWalker).

Рисунок 8-5 показывает типичный экран отладки Perl. Обратите внимание, что мы добавили мое ключевое слово для глобальных переменных.

8.3 Питон

Давайте возьмем в качестве примера tf.py, который подсчитывает слова, строки и абзацы в текстовый файл, как в нашем примере Perl выше.

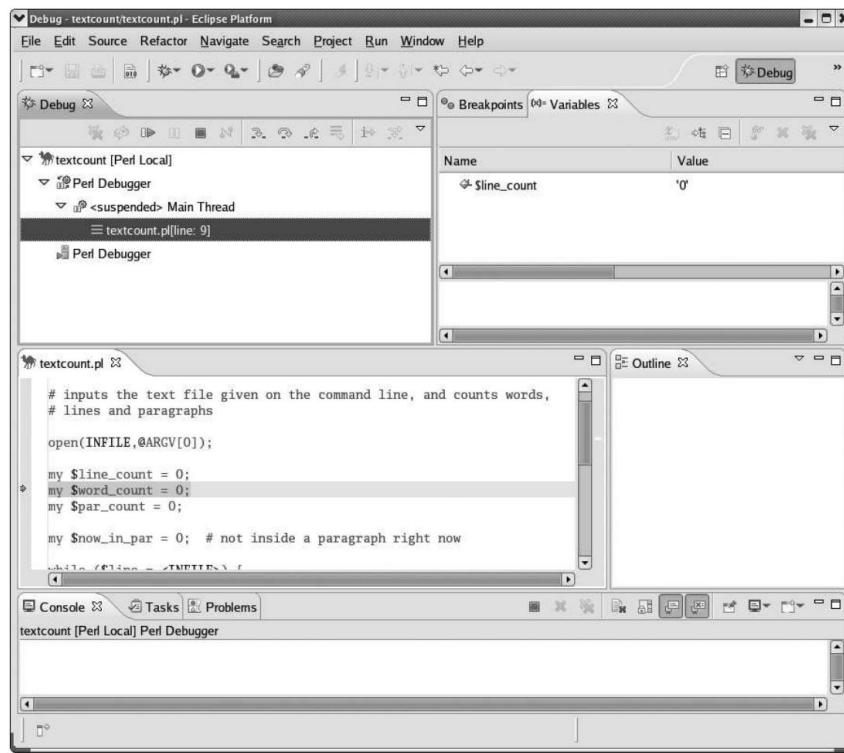


Рисунок8-5: Отладка Perl экран

Текстовый файл 1 класса:

```

2      nfiles = 0 # количество объектов текстового файла
3      def __init__(self,fname):
4          текстовыйфайл.nfiles += 1
5          self.name = fname # имя
6
7          self.fh = открыть(имя_ф)
8          self.nlines = 0 # количество строк
9          self.nwords = 0 # количество слов
10         self.npars = 0 # количество слов
11         self.lines = self.fh.readlines()
12         self.wordlineparcount()
13
14     def wordlineparcount(self):
15         "находит количество строк и слов в файле"
16
17         self.nlines = len(self.lines)
18         вабзац = 0
19         для l в self.lines:
20
21             w = l.split()
22             self.nwords += len(w)
23             если l == '\n':

```

```

20         если впункте:
21             вабзац = 0
22         elif не в абзаце:
23             self.npars += 1
24             вабзац = 1
25
26     def grep(сам, цель):
27         "выводит все строки в файле, содержащие целевой объект"
28         для l в self.lines:
29             если l.find(цель) >= 0:
30                 распечатать l
31             распечатать я
32
33 определение main():
34     t = текстовый_файл('test.txt')
35     распечатать t.nwords, t.nlines, t.npars
36
37 если __name__ == '__main__':
38     основной()

```

Обязательно включите в исходный код две строки, показанные в конце программы:

```

если __name__ == '__main__':
    основной()

```

ПРИМЕЧАНИЕ Если вы опытный программист Python, вы, вероятно, хорошо знаете этот шаблон. Если вы не знакомы с Python, краткое объяснение этих строк заключается в том, что вы необходимо проверить, вызывается ли ваша программа Python сама по себе или импортируется как модуль в какую-то другую программу. Это проблема, которая возникает при запуске программы через отладчик.

8.3.1 Отладка Питон^В ДДД

Базовый отладчик Python — это PDB (pdb.py), текстовый инструмент. Его полезность значительно улучшено за счет использования DDD в качестве графического интерфейса пользователя.

Однако, прежде чем начать, нужно уладить некоторые дела. Чтобы DDD мог правильно обращаться к PDB, Ричард Вольф написал PYDB (pydb.py), слегка измененная версия PDB. Затем вы запустите DDD с опция -pydb . Но Python эволюционировал, и оригинальный PYDB перестал работать правильно.

Хорошее решение было разработано Рокки Бернстайном. На момент написания этой статьи, летом 2007 года, его модифицированный (и значительно расширенный) PYDB должен был быть включены в следующий релиз DDD. В качестве альтернативы вы можете использовать патч которые Ричард Вольф любезно предоставил. Вам понадобятся следующие файлы:

- <http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydb.py>
- <http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbcmd.py>

- <http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbsupt.py>

Поместите файлы в какой-нибудь каталог, например /usr/bin, дайте им права на выполнение.
создайте файл с именем pydb , состоящий из одной строки

`/usr/bin/pydb.py`

Убедитесь, что вы также даете файлу pydb разрешение на выполнение. После этого
сделано, вы можете использовать DDD для программ Python. Затем запустите DDD:

`$ DDD --pydb`

ПРИМЕЧАНИЕ. Если появляется окно с сообщением «PYDB не удалось запустить», возможно, у вас есть путь
проблемы — например, у вас может быть другой файл с именем pydb. Просто убедитесь, что
один для DDD будет встречен первым в вашем пути поиска.

Затем добавьте исходный файл tf.py, выбрав Файл | Открыть исходный код
и дважды щелкнув имя файла или введя

`файл tf.py`

в консоли. Запрос (Pdb) может не отображаться изначально, но введите свой
командовать в любом случае.

Ваш исходный код появится в окне «Исходный текст», и вы сможете
затем установите точки останова как обычно. Предположим, что вы установили точку останова на строке

`w = l.split()`

Чтобы запустить программу, нажмите Программа | Выполнить, задайте любые аргументы
командной строки во всплывающем окне Выполнить, а затем нажмите Выполнить. После нажатия Выполнить,
вам нужно будет дважды нажать «Продолжить». Это связано с работой базового отладчика PDB/
PYDB. (Если вы забудете это сделать, PDB/PYDB напомнит вам об этом в консоли DDD.)

Если вы вносите изменения в исходный код, выполните команду file в
Консоль, или выберите Источник | Перезагрузить источник. Ваши точки останова с последнего
Прогон будет сохранен.

8.3.2 Отладка Питон в Eclipse

Опять же, процедуры для Python похожи на процедуры для других языков. Это, конечно, иллюстрирует
ценность наличия единого инструмента для использования на
несколько языков — как только вы научитесь использовать инструмент на одном языке,
его довольно легко использовать для другого. Основные Python-специфичные точки
имейте в виду:

- Вам необходимо установить плагин Pydev. После этого выберите Окно |
Настройки | Pydev и сообщите Eclipse о местоположении вашего Python
интерпретатора, например, /usr/bin/python.

- Используйте Pydev Package Explorer в качестве навигатора.
- Не создаются постоянные файлы байт-кода, поэтому нет процедуры сборки. дюре.
- При настройке диалогового окна запуска/отладки обратите внимание на следующее:
 - заполните поле Main Module именем исходного файла, в котором вы хотите начать выполнение.
 - На вкладке «Аргументы» заполните поле «Базовый каталог», чтобы указать тот, в котором вы находитесь.
 - есть входные файлы (или корень дерева таких каталогов), и поместите свой Аргументы командной строки в Аргументах программы.
- В перспективе отладки значения переменных доступны только в представлении «Переменные» и только для локальных переменных.

Одним из главных преимуществ Eclipse перед DDD является то, что PDB, базовый механизм отладки, используемый DDD, не работает с потоковыми программами, тогда как Eclipse это делает.

8.4 Отладка кода SWIG

SWIG (упрощенная оболочка и генератор интерфейсов) — популярный открытый Исходный инструмент для сопряжения Java, Perl, Python и ряда других интерпретируемых языков с C/C++. Он включен в большинство дистрибутивов Linux и также может быть загружен из Интернета. Он позволяет вам написать большую часть код приложения на интерпретируемом языке и включить в него определенные разделы, написанные вами, например, на C/C++, для повышения производительности.

Возникает вопрос, как запустить GDB/DDD на таком коде. Здесь мы представим небольшой пример с использованием Python и С. Код С будет управлять Очередь «первым пришел — первым ушел» (FIFO):

```

1 // fifo.c, пример SWIG; управляет очередью символов FIFO
2
3 символа *fifo; // очередь
4
5 int nfifo = 0, // текущая длина очереди
6     maxfifo; // максимальная длина очереди
7
8 int fifoinit(int spfifo) // выделить место для максимального количества элементов spfifo
9 { fifo = malloc(spfifo);
10    если (fifo == 0) вернуть 0; // неудача
11    еще {
12        максифо = спфиго;
13        возврат 1; // успех
14    }
15 }
16
17 int fifoput(char c) // добавить с в очередь
18 { если (nfifo < maxfifo) {
19     фифо[нфиго] = с;
20     возврат 1; // успех

```

```

21      }
22      иначе вернуть 0; // неудача
23 }

24
25 символов fifoget() // удалить начало очереди и вернуться
26 { символ c;
27 если (nfifo > 0) {
28     c = фифо[0];
29     memmove(fifo,fifo+1,--nfifo);
30     возврат c;
31 }
32 в противном случае вернуть 0; // предполагаем, что в очереди нет нулевых символов
33 }

```

Помимо файла .c , SWIG также требует файл интерфейса, в данном случае fifo.i. Он состоит из глобальных символов, перечисленных один раз в стиле SWIG и один раз в стиле C:

```

%модуль fifo

%{extern char *fifo;
внешний int nfifo,
    максфибо;
extern int fifoinit(int);
extern int fifoput(char);
внешний символ fifoget(); %}

внешний символ *fifo;
внешний int nfifo,
    максфибо;
extern int fifoinit(int);
extern int fifoput(char);
внешний символ fifoget();

```

Чтобы скомпилировать код, сначала запустите swig, который сгенерирует дополнительный файл .c и файл Python. Затем используйте GCC и Id для создания динамического общего объекта .so библиотека. Вот Makefile:

```

_fifo.so: fifo.o fifo_wrap.o
    gcc -shared fifo.o fifo_wrap.o -o _fifo.so

fifo.o fifo_wrap.o: fifo.c fifo_wrap.c
    gcc -fPIC -g -c fifo.c fifo_wrap.c -I/usr/include/python2.4

fifo.py fifo_wrap.c: fifo.i
    swig -python fifo.i

```

Библиотека импортируется в Python как модуль, как показано в тестовой программе:

```
# testfifo.py

импорт fifo

def main():
    fifo.fifoinit(100)
    fifo.fifoput('x')
    fifo.fifoput('c') с =
    fifo fifoget() печать cc
    =
    fifo fifoget() печать с

если __name__ == '__main__': main()
```

Вывод этой программы должен быть «х» и «с», но мы ничего не получаем:

```
$ python testfifo.py
```

```
$
```

Чтобы использовать GDB, помните, что фактическая программа, которую вы запускаете, — это интерпретатор Python, `python`. Поэтому запустите GDB на интерпретаторе:

```
$ gdb питон
```

Теперь мы хотели бы установить точку останова в присоединенной библиотеке FIFO, но библиотека еще не загружена. Загрузка не произойдет, пока не будет импортирована строка

```
импорт fifo
```

выполняется интерпретатором Python. К счастью, мы можем попросить GDB остановиться на функции в библиотеке в любом случае:

```
(gdb) b fifoput
```

Функция "fifoput" не определена.

Сделать точку останова ожидающей при будущей загрузке общей библиотеки? (у или [п]) у

Ожидается точка останова 1 (fifoput).

Теперь запустите интерпретатор, аргументом которого является тестовая программа `testfifo.py`:

```
(gdb) r testfifo.py Запуск
программы: /usr/bin/python testfifo.py Чтение символов из
общего объекта, считанного из целевой памяти...(отладочные символы не найдены)...выполнено.

Загруженная система предоставила DSO по адресу
0x164000 (отладочные символы не
найдены) (отладочные символы не
найдены) (отладочные символы не найдены)
[Отладка потоков с использованием libthread_db включена]
[Новая тема -1208383808 (LWP 15912)]
(отладочные символы не найдены)
(отладочные символы не найдены)
(отладочные символы не найдены)
(отладочные символы не найдены)
Точка останова 2 по адресу 0x3b25f8: файл fifo.c, строка 18.
Ожидаемая точка останова "fifoput" разрешена
[Переключение на поток -1208383808 (LWP 15912)]
```

Точка останова 2, fifoput (c=120 'x') в fifo.c:18 { if (nfifo < maxfifo) { 18

Теперь вы можете делать то, что вы уже так хорошо знаете:

```
(гдб) н 19
фифо[nfifo] = c;
(gdb) p nfifo $1 =
0 (гдб)
с
Продолжение.
```

Точка останова 2, fifoput (c=99 'c') в fifo.c:18 { if (nfifo < maxfifo) { 18

```
(гдб) н
19           fifo[nfifo] = c; (гдб) p
nfifo $2 = 0
```

Ну, вот в чем проблема. Каждый раз, когда вы пытаетесь добавить символ в очередь,
вы просто перезаписываете ранее добавленный символ. Стока 19 должна быть

```
фифо[nfifo++] = c;
```

После внесения этих изменений код будет работать нормально.

8.5 Язык ассемблера

GDB и DDD могут быть чрезвычайно полезны при отладке языка ассемблера. код. Есть ряд особых соображений, которые следует иметь в виду, будут описаны в этом разделе.

В качестве примера возьмем код в файле testff.s:

```

1 # подпрограмма findfirst(v,w,b) находит первое вхождение значения v
2 # в блоке из w последовательных слов памяти, начиная с b, возвращая
3 # либо индекс слова, где была найдена буква v (0, 1, 2, ...), либо -1, если
4 # v не найден; начиная с _start, у нас есть короткий тест
5 # подпрограмма
6
7 .data # сегмент данных
8 x:
9     .длинный 1
10    .длинный 5
11    .длинный 3
12    .длинный 168
13    .длинный 8888
14 .text # сегмент кода
15 .globl _start # требуется
16 _start: # требуется использовать эту метку, если не предприняты специальные действия
17     # поместить аргументы в стек, затем выполнить вызов
18     # нажмите $x+4 # начните поиск с 5
19     # нажмите $168 # поиск 168 (намеренно не по порядку)
20     # нажмите $4 # поиск 4 слов
21     # позвоните findfirst
22 сделано:
23     movl %edi, %edi # фиктивная инструкция для точки останова
24 найти первым:
25     # находит первое вхождение указанного значения в блоке слов
26     # EBX будет содержать искомое значение
27     # ECX будет содержать количество слов для поиска
28     # EAX укажет на текущее слово для поиска
29     # возвращаемое значение (EAX) будет индексом найденного слова (-1, если не найдено)
30     # извлечь аргументы из стека
31     movl 4(%esp), %ebx
32     movl 8(%esp), %ecx
33     movl 12(%esp), %eax
34     movl %eax, %edx # сохранить начальное местоположение блока
35     # начало цикла; сравнить текущее слово с искомым значением
36 верх: cmpl (%eax), %ebx
37     найдено jz
38     decl %ecx # уменьшить счетчик количества слов, оставшихся для поиска
39     jz notthere # если счетчик достиг 0, искомого значения нет
40     addl $4, %eax # в противном случае перейти к следующему слову
41     jmp топ

```

```

Найдено 42 :

43      subl %edx, %eax # получить смещение от начала блока
44      shr $2, %eax # разделить на 4, чтобы преобразовать из смещения байта в индекс
45      в отставке
46 нет:
47      movl $-1, %eax
48      в отставке

```

Это язык ассемблера Linux Intel, использующий синтаксис AT&T, но пользователи знакомые с другими синтаксисами Intel, вы легко поймете код. (Команда GDB устанавливает disassembly-flavor intel, что заставит GDB отобразить весь вывод своей команды disassemble в синтаксисе Intel, который похож на синтаксис, используемый (Например, компилятором NASM. Кстати, поскольку это платформа Linux, программа работает в 32-битном плоском режиме процессора Intel.)

Как указано в комментариях, подпрограмма findfirst находит первый вхождение указанного значения в указанный блок последовательных слов памяти. Возвращаемое значение подпрограммы — это индекс (0, 1, 2, ...) слова, в котором найдено значение, или -1, если оно не найдено.

Подпрограмма ожидает, что аргументы будут помещены в стек, так что при входе стек выглядит так:

адрес начала блока для поиска
количество слов в блоке
значение для поиска
обратный адрес

ПРИМЕЧАНИЕ. Стеки Intel растут вниз, то есть к адресу 0 в памяти. Слова с меньшим адреса показаны ниже на рисунке.

Чтобы ввести ошибку, которую мы можем найти с помощью GDB, мы намеренно смешивали элементы в последовательности вызова в «основной» программе:

нажмите \$x+4 # начните поиск с 5
нажмите \$168 # поиск 168 (намеренно не по порядку)
нажмите \$4 # поиск 4 слов

Вместо этого инструкции, предшествующие вызову, должны быть

нажмите \$x+4 # начните поиск с 5
нажмите \$4 # поиск 4 слов
нажмите \$168 # поиск 168

Точно так же, как вы используете опцию -g при компиляции кода C/C++ для использования с GDB/DDD, здесь на уровне сборки вы используете -gstabs:

\$ как -a --gstabs -o testff.o testff.s

Это создает объектный файл testff.o и выводит параллельное сравнение исходного кода сборки и соответствующего машинного кода. Он также показывает смещения элементов данных и другую информацию, которая потенциально полезна для процесса отладки.

Затем мы связываем:

```
$ ld testff.o
```

В результате создается исполняемый файл с именем по умолчанию a.out. Давайте запустим этот код в GDB:

```
(gdb) b done
Точка останова 1 по адресу 0x8048085: файл testff.s, строка 18. (gdb) r
Запуск
программы: /debug/a.out Точка останова 1,
done () по адресу testff.s:18 18 movl %edi, %edi #
фиктивная точка останова
Текущий язык: авто; в настоящее время asm
(gdb) p $eax $1
= -1
```

Как вы можете видеть здесь, к регистрам можно обращаться через префиксы знака доллара, в этом случае \$eax для регистра EAX. К сожалению, значение в этом регистре равно -1, что указывает на то, что искомое значение, 168, не было найдено в указанном блоке.

При отладке программ на языке ассемблера одним из первых действий является проверка стека на точность. Итак, давайте установим точку останова в подпрограмме, а затем проверим стек, когда доберетесь туда:

```
(gdb) b findfirst
Breakpoint 2 по адресу 0x8048087: файл testff.s, строка 25.
(gdb) r
Отлаживаемая программа уже запущена.
Начать с начала? (y или n) у Запуск программы: /
debug/a.out Точка останова 2,
findfirst () в testff.s:25 movl 4(%esp), %ebx
25
(gdb) x/4w $esp
0xbffffd9a0: 0x08048085 0x00000004 0x000000a8 0x080490b4
```

Стек, конечно, является частью памяти, поэтому для его проверки вы должны использовать команду GDB x , которая проверяет память. Здесь мы попросили GDB отобразить четыре слова, начиная с места, указанного указателем стека ESP (обратите внимание, что на рисунке стека выше показаны четыре слова). Команда x отобразит память в порядке возрастания адресов. Это именно то, что вам нужно, поскольку на архитектуре Intel, как и на многих других, стек растет в сторону 0.

Из рисунка стека, показанного выше, видно, что первое слово должно быть адресом возврата. Это ожидание можно проверить разными способами. Одним из подходов было бы использование команды GDB `disassemble`, которая перечисляет инструкции языка ассемблера (обратно переведенные из машинного кода) и их адреса. Войдя в подпрограмму, вы могли бы затем проверить, соответствует ли содержимое первого слова в стеке адресу функции, которая следует за вызовом.

Вы увидите, что это так. Однако вы обнаружите, что второе число, 4, которое должно быть значением для поиска (168), на самом деле является размером блока поиска (4). Из этой информации вы быстро поймете, что мы случайно поменяли местами две инструкции `push` перед вызовом.

ИНДЕКС

A

ошибки доступа, определены, 223 адреса, см. также адреса виртуальной памяти
контрольные точки, 68 массивов, см. также динамические массивы, проверка, 37 искусственных массивов
ДДД, 107
GDB, 106
язык ассемблера, GDB и DDD, 255–258 атрибуты, точки останова, 67–69

Б

обратная трассировка, сегментация местоположение ошибки, 4 двоичный поиск принцип синтаксические ошибки, 189 использование, 4 двоичные деревья, пример, 95 скобки, сопоставление, 208 точки останова, 47–94, см. также аппаратно-поддерживаемые точки останова; временные точки останова около, 47 списки команд, 85–89 условные, 79–84 DDD, 9 удаление, отключение и перемещение, 60–66 Пример GDB, 56 списки, 49 сохранение, 59 возобновление выполнения, 69–79 настройка, 51–56 использование, 14 просмотр атрибутов, 67–69 точки наблюдения, 89–94 ошибки, ошибки доступа к памяти и сегментации, 124 скрипты сборки, библиотеки, 193 ошибки шины, 128

С

язык C, см. также отчет об ошибках библиотеки GNU C, 213–217 Расширения библиотеки C99, splint, 221 команда вызова, 100 стеков вызовов, перемещение вверх и вниз, определено 17 точек захвата, 55 ГБД, 48 CGDB, 13 клиент-серверных сетевых программ, методы мультипрограммирования, 145–151 список команд, точки останова, 85–89 команд, см. также конкретные команды ДДД, 161 Затмение, 161 GDB, 161 команда команда, 99 компиляторов используют, 212 предупреждений, 210 компилируют, 185–193 отсутствующих библиотек, 190 многофайловых программ, 57 фантомных номеров строк в сообщениях об ошибках си условные точки останова Eclipse, 42 использование, 79–84 условные выражения, точки наблюдения, 17 сценариев настройки, 193 принцип подтверждения о, 2 ГБД, 25, 31 Java, 239 возобновлений выполнения, 69 сообщений об ошибках синтаксиса, 187 команд продолжения, 74 вспомогательные переменные имена, 115

использование, 114
 сбои основных файлов, 129–131
 местоположение сбоя
 сегмента, 4 сбоя, 117–144 основные файлы, 129–131 пример, 131–144 управление памятью, 118–129 критические секции, использование, 156 программы curses, 194–201

Д
 раздел данных, 119
 DDD (Data Display Debugger) about, 2 языка ассемблера, 255–258 списков точек останова, 50 сводок команд, 161 условная точка останова, 83 удаления и отключения точек останова, 62 против Eclipse, 9 пример, 36–37 проверка переменных, 102 Java, 240–241 подвижные точки останова, 64 Perl, 244 Python, 249 возобновление выполнения, 78 установка точек останова, 55 установка переменных, 113 отмена и повтор действий точек останова, 66 использование графического интерфейса, 6 просмотр атрибутов точек останова, 69 значения по умолчанию, обработчики сигналов, 126 удаление точек останова, 60–64 Алгоритм Дейкстры, 172 отключение точек останова, 62–64 команда отображения, 98, 102 отображение переменных, 112 размещение, точки останова, 68 двойных освобождений определено, 224 mtrace(), 231 дамп ядра, 129, 130 динамические массивы, проверка, 104 динамически выделенная память, 221–233

обнаружение проблем, 224 Библиотека Electric Fence, 225–228 Инструменты библиотеки GNU C, 228–233

Э

Eclipse about, 2 списка точек останова, 51 сводка команд, 161 условная точка останова, 84 против DDD, 9 удаление и отключение точек останова, 63 пример, 38–43 проверка переменных, 103 Perl, 246–247 Python, 250 возобновление выполнения, 79 установка точек останова, 56 установка переменных, 113 использование графического интерфейса, 7 просмотр атрибутов точек останова, 69 Библиотека Electric Fence, динамически выделяемая память, 225–228 Emacs, функции, 206 включение, точки останова, 62 errno, использование, 213–217 ошибки, см. также ошибки доступа; ошибки шины; двойное освобождение; предупреждения параметры компилятора для проверки, 213 отчеты на языке C, 213–217 примеры, см. примеры программ исключения, сбои, 128 выполнение, возобновление, 69–79 выражения, см. также условные выражения библиотечные функции, 82 точки наблюдения, 92, 93

Ф

ошибки, см. ошибки сегментации Очередь FIFO, пример, 251 файл, см. основные файлы; файлы запуска компиляция многофайловых программ, 57 команда завершения, 74 FPE (исключение с плавающей точкой), 128 функций, см. также специальные функции обработки строк, 132

Г

GDB (отладчик проекта GNU), см. также
CGDB
about, 2
benefits, 11 assembler
language, 255–258 breakpoint lists, 49
command summary, 161
conditional breakpoints, 80–83
deleting breakpoints, 61 disabling
breakpoints, 62 example, 22–36
expressions, 93 inspecting variables,
98 Java, 238–241
resuming execution,
70–78 setting breakpoints, 51
setting variables,
113 startup files, 43 TUI mode, 12 viewing
breakpoint attributes, 67 .gbdinit
files, 43 general protection
fault, see segmentation
fault GNU C library
dynamically allocate memory, 228–233 GNU
Project Debugger, see
GDB GNU/Linux, see Linux guard
statements, using, 156 GUI
programs, 194–201
GUI-based tools, see also DDD (Data Display
Debugger);
Предимущества Eclipse, 10 по сравнению с
текстовыми инструментами, 5–
14 использование DDD в качестве
графического интерфейса для
JDB, 241

час

аппаратные точки наблюдения, 91
аппаратно-поддерживаемая точка останова,
определенены, 54
справка, см. онлайн-справку

я

IDE (интегрированная среда разработки),
текстовые редакторы как, 211

идентификаторы, контрольные точки, 49

бесконечные циклы

GDB, 27

прерываний, 4

проверки переменных, 15

установка

DDD, 2

Eclipse, 2

интегрированные среды разработки

(IDE), текстовые редакторы как, 211

Стеки Intel, 256 интерфейсов,

текстовый интерфейс по сравнению с графическим пользовательским
интерфейсом, 5–14 прерываний, бесконечные циклы, 4

жк

Java, с использованием GDB, DDD и Eclipse, 236–242

JDB (java Debugger), DDD как графический интерфейс пользователя
для, 241

л

макет, см. макет программы

лексическая подсветка, определено, 206

библиотек, см. также расширения

библиотек C99; программы curses;

Библиотека Electric Fence; библиотека GNU
C; отсутствует компиляция

статических библиотек, 190

Архитектура NOW, 170

SDSM, 170

типов, 191

вызовов библиотек, errno, 213

библиотечных функций

в сравнении с системными вызовами, 217

Выражения GDB, 82 номера

строк, см. линтинг фантомных номеров
строк,

использование, 219–221

Linux, дамп ядра, 130 списков, см.

также списки команд точки останова,

49 загрузка, 185–193

отсутствующие

библиотеки, 190 фантомные

номера строк в сообщениях об ошибках

синтаксиса, 185–190 локальные

переменные, мониторинг, 112 циклы, см.

бесконечные циклы ltrace,

использование, 217–219

M

make-файлы

и предупреждения компилятора, 210
и Vim, 209

MALLOC_CHECK_, 228 mcheck(), 230

память, см. также

динамически выделяемая память; виртуальное
адресное пространство; непосредственная
проверка адресов виртуальной
памяти, 112 утечки памяти,
mtrace, 231 управление памятью,
сбои, 118–129

передача сообщений, определено, 163

системы передачи сообщений,
методы
многопрограммирования,
164–169 модульный подход, см. мониторинг
подхода
«сверху вниз», локальные переменные, 112
перемещение, точки останова в DDD, 64
mtrace(), 231
методы многопрограммирования, 145–183

клиент/сервер сетевые программы, 145–151
пример,
171–183 параллельные
приложения, 163–171 потоковый код,
151–163 muntrace(), 231

N

сети, методы многопрограммирования
для клиент-серверных сетевых
программ, 145–151 нецелочисленные
возвращающие функции, 83
Архитектурные библиотеки

NOW, 170

O

смещения, GDB, 53

онлайн-справка
о, 19

Пример

OpenMP, 171–183
настоящая разделяемая память,
170 операций, 14–18
проверка переменных, 15

перемещение вверх и вниз по стекам вызовов,
17 пошаговое выполнение исходного кода,
14 точек наблюдения, 17

P

страницы таблиц, 122

страниц

о, 121

Системы SDMS, 170

параллельных приложений,

методы

многопрограммирования,

163–171 скобки, балансировка, 208

Perl, DDD и Eclipse, 242–247 perror(), 216

сохранение, точки

останова, 59 фантомные номера строк,

сообщения об ошибках синтаксиса, 185–190

программа pkconfig, 193

простой текст, см. принципы

текста, 2–5, см. также

принцип бинарного поиска; принцип

подтверждения; подтверждение

подхода сверху вниз, 2 другое, 4 printf(),

использование с

кодом

трассировки, 3 печать, переменные, 112

таблицы процессов, определено,

152 процессы, определено, 152 макет

программы, память, 118

Pthreads, пример, 151

Python, DDD и Eclipse, 247–251

R

повтор действий точек останова в DDD, 66 отчетов,
просмотр ошибок; предупреждения

C

примеры программ

проверки и установки переменных, 95, 109

Вводный сеанс отладки, 19–43

ошибки сегмента, 131–144, 171–183

установка точек останова с помощью GDB, 56

потоковый код, 153–161

сохранение таблиц символов, 21

SDSM (программное обеспечение распределенной общей памяти), библиотеки, 170

основных файлов ошибок сегментации, 131 определенный, 118 определение местоположения, 4

Затмение, 42

GDB, 32

ошибки доступа к памяти, 124

сигналы Unix, 125

установка точек останова, 51–56 переменные, 113 точки наблюдения, 90 общая память, определено, 163 пример систем с общей памятью, 171–183 методы многопрограммирования, 170 оболочки, файлы ядра, 130 Решето Эратосфена, 153 обработчика сигналов, сигналы Unix, 126 сигналов, ошибки сегментации, 125 Упрощенная оболочка и интерфейс Генератор (SWIG), с использованием, 251–254

snprintf(), 143 сокета, использование, 148

программно распределенная разделяемая память (SDSM), библиотеки, 170

исходный код, пошаговое выполнение, 14 splint, использование, 220–221 кадры стека, 17 разделы стека, 119 стеки, см. стеки вызовов; Intel стекирует файлы запуска, использование, 43–45 статические средства проверки кода, lint и другие инструменты, 219–221 статические библиотеки, использование, 191 пошаговое выполнение в сравнении с обходом функции, 72 исходный код, 14 strace, использование, 217–219 strerror(), 216 обработка строк, функции, 132 SWIG (упрощенная оболочка и Интерфейсный генератор), с использованием, 251–254 переключателей, шплинт, 221

таблицы символов, сохранение, 21 сообщение об ошибках синтаксиса, фантомные номера строк, 185–190 подсветка синтаксиса, текстовые редакторы, 206 системных вызовов, сравнение с библиотечными функциями, 217

Т

таблицы, см. таблицы процессов; таблицы символов задачи, см. процессы временные точки останова определены, 52 Eclipse, 56 текстовых редакторов, 206–212 как IDE, 211 makefiles и предупреждения компилятора, 210

соответствующих брacketов, 208 подсветка синтаксиса, 206 Vim и makefiles, 209 текстовый раздел, 118 текст, GDB, 6 против инструментов на основе графического интерфейса, 5–14 веток о, 151 определено, 153 методы многопрограммирования, 151–163 подход сверху вниз о, 4

GDB, 29 шагов, 73 кода трассировки, использование, 3 дерева, см. двоичные деревья Режим TUI, GDB, 12

У

команда ulimit, 131 отмена действий точек останова в DDD, 66

Сигналы Unix и ошибки сегментации, 125 адресов виртуальной памяти, 118 До команды, 75

В

история значений, использование, 114

переменные, 95–115, см. также переменные удобства;

локальные переменные,

исследующие память напрямую, 112 пример, 95,

109

Собственность GDB, 113

проверка, 15 печать

и отображение, 112 настройка,

113 контрольные

точки, 17

Книга

о Vim, 212 вызов

make, 210 и makefiles,

209 подсветка

синтаксиса, 206 виртуальное

адресное пространство, страницы, 121

виртуальная память адреса

точки останова в GDB, 53

Юникс, 118

ВТ

предупреждения, см. также ошибки

параметры компилятора для, 212

компиляторы и makefiles, 210 splint, 220

watchpoints, см.

также hardware watchpoints binary search, 5

GDB, 48 с

использованием, 17, 89–94



**FREE SOFTWARE
FOUNDATION**

Since 1985 we've been fighting for essential freedoms for computer users



Supporting freedom

- Find careers in free software.
- Get help with free software licensing.
- Find hardware compatible with free software.
- Get connected — stay informed.

www.fsf.org

Join the FSF as an associate member or corporate patron today.

ОБНОВЛЕНИЯ

Посетите <http://www.nostarch.com/debugging.htm> для получения обновлений, сведений об ошибках и другой информации.

КОЛОФОН

Шрифты, использованные в The Art of Debugging, — New Baskerville, Futura, The Sans Mono Condensed и Dogma. Книга была набрана с помощью пакета LATEX 2 ϵ nostarch Борисом Вейцманом (2008/06/06 v1.3 Набор книг для No Starch Press).