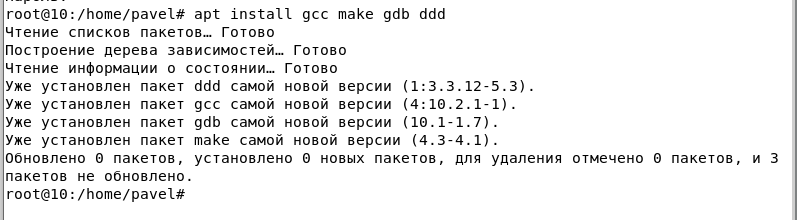
Отчет по лабораторной работе:

**Компилятор GCC. Отладчик GDB. Отладчик DDD**

**Выполнил:**

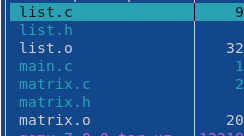
**Студент группы КФ-17 Гуревич П.А.**

**Задание 1.Установите пакеты gcc, make, gdb, ddd.**

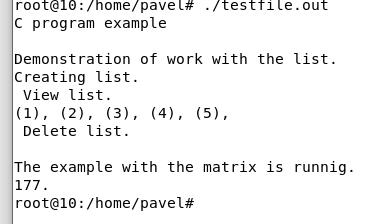


**Задание 2. Создайте на компьютере файлы исходного кода программ на языке программирования C:**

**main.c, list.h , list.c, matrix.h, matrix.c**.



**Задание 3. Выполните компиляцию единого исполняемого файла из трех файлов исходного кода. При компиляции нужно указать, что используется библиотека libm.a. Назовите полученный в результате исполняемый файл testfile.out и убедитесь в его работоспособности.**



**Задание 4. Выполните компиляцию статической библиотеки teststatic.a, которая содержит исходный код из файлов lists.c и matrix.c.**



**Задание 5.** **Выполните компиляцию разделяемой библиотеки testdynamic.so, которая содержит исходный код из файлов lists.c и matrix.c.**





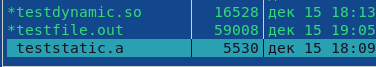
**Задание 6**. **Выполните компиляцию исполняемого файла в соответствии с пунктом 3 по очереди со следующими параметрами оптимизации: «-O1», «-O2», «-O3», «-Os». Сравните результаты работы и размер полученных файлов.**

**С параметрам оптимизации: -O1 ничего не поменялось.**

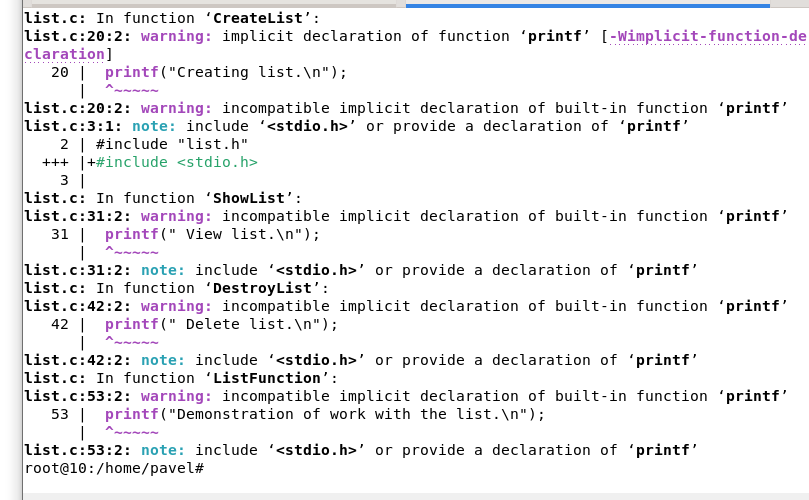
**С параметрам оптимизации: -O2» ничего не поменялось.**

**С параметрам оптимизации: -O3 ничего не поменялось.**

**С параметрам оптимизации: -Os размер файла увеличился.**

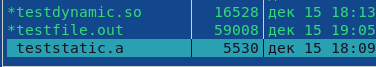


**Задание 7.** **Выполните компиляцию исполняемого файла в соответствии с пунктом 3 с включением всех предупреждений компилятора. Проанализируйте выдаваемые предупреждения.**

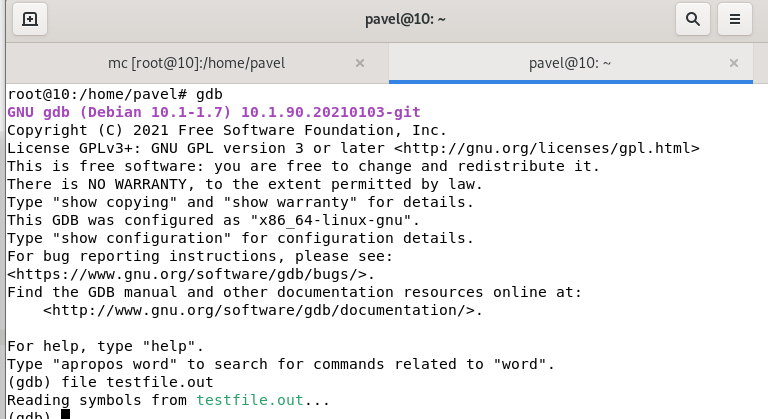


**Задание 8.Выполните компиляцию исполняемого файла в соответствии с пунктом 5 с включением в исполняемый файл отладочной информации и без оптимизации.**



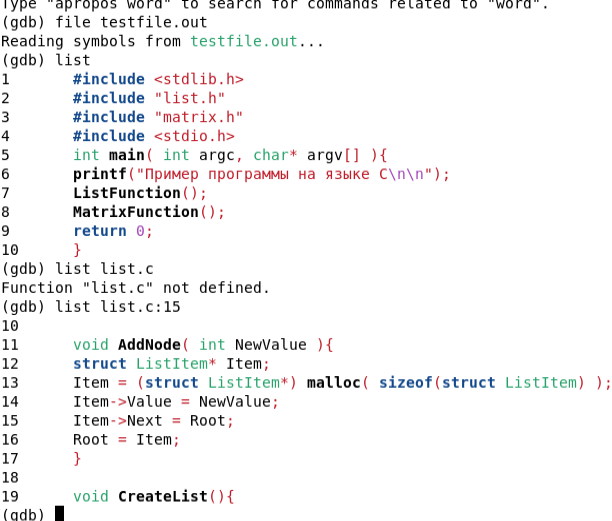


**Задание 9.** **Выполним отладку файла, полученного в задании 8 с помощью отладчика gdb. Запустите отладчик gdb одноименной командой. При этом на экране отобразится информация о версии программы и авторских правах, а также приглашение ко вводу встроенных команд:**

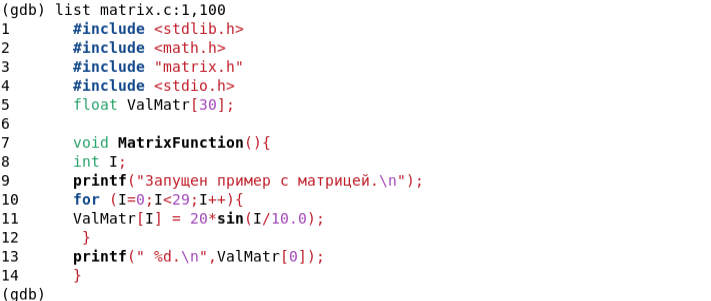


**Задание 10.** **Просмотрите текст программы с помощью команды list:**

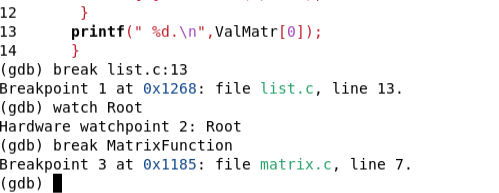
**В результате на экране отобразится 10 ближайших к указанной строке строк заданного файла. Например: (gdb) list list.c:15**



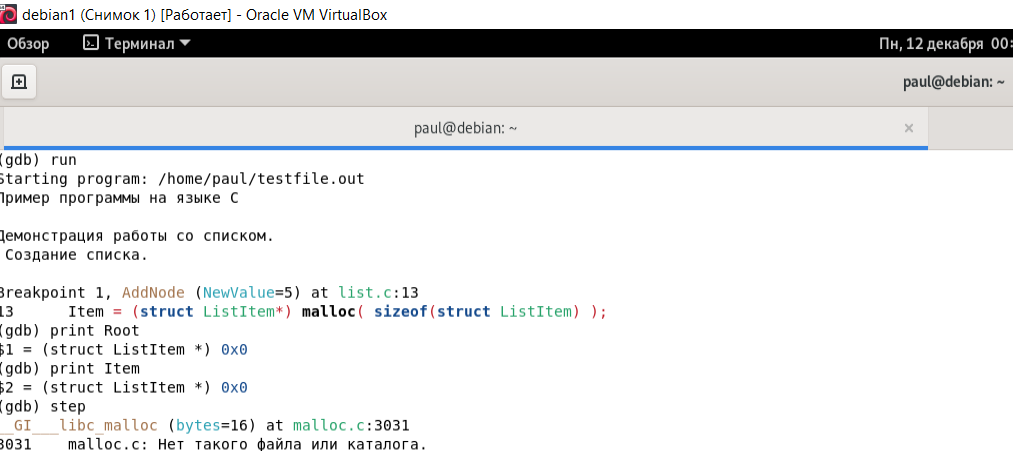
**Используя данную команду, отобразите весь исходный код файлов list.c и matrix.c:**



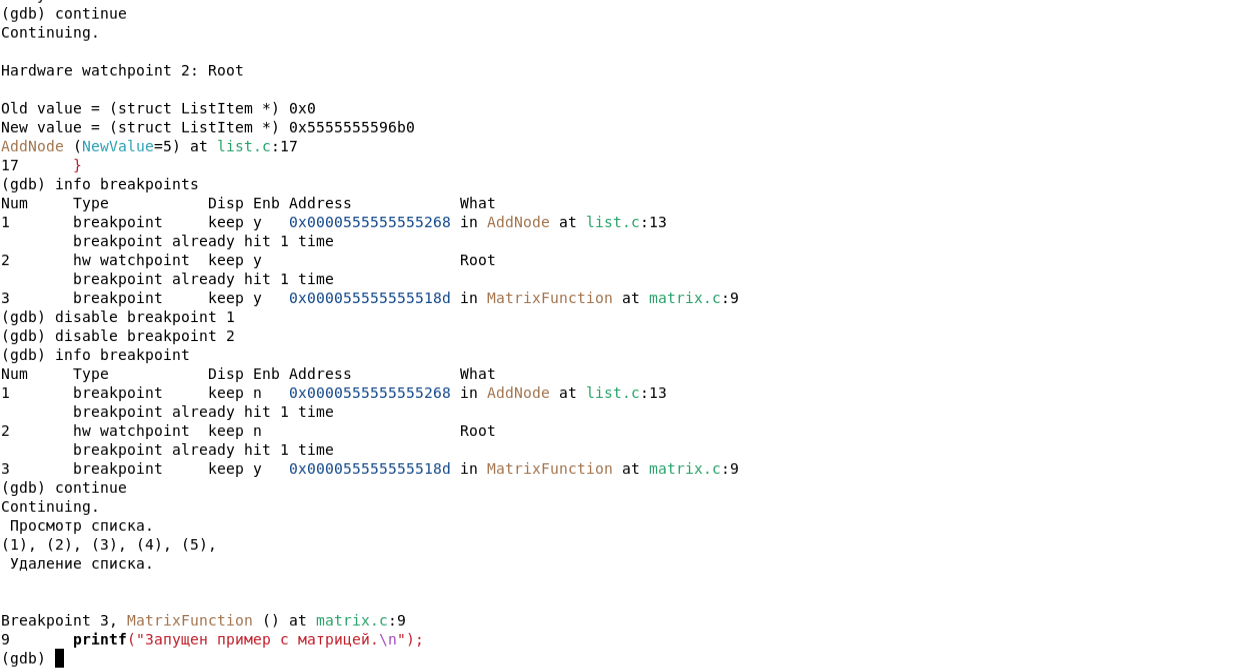
**Задание 11. Перед запуском программы на выполнение, определим несколько точек останова. Первую точку останова поставим на первый исполняемый оператор функции AddNode() в файле list.c,** **второй поставим аппаратную точку останова при изменении переменной Root, третью точку останова свяжем с запуском функции MatrixFunction():**



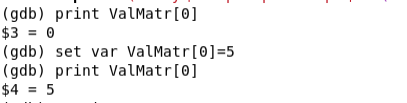
**Задание 12.Выполним пошаговую отладку программы с использованием определенных на предыдущем шаге выполнения точек останова.**



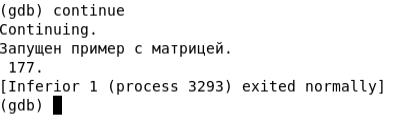
Теперь мы остановились по аппаратной точке останова при изменении переменной Root. Остановка выполнения программы произошла в 17-й строке файла list.c (это функция AddNode()). Просмотрим состояние точек останова и статистику по ним. Отключим две первые точки останова и опять просмотрим состояние точек останова. Продолжим выполнение программы:



Теперь мы остановились в третьей точке останова в функции MatrixFunction. Продемонстрируем возможности отладчика по работе с переменными. Просмотрим, а потом изменим значение нулевого элемента массива ValMatr:



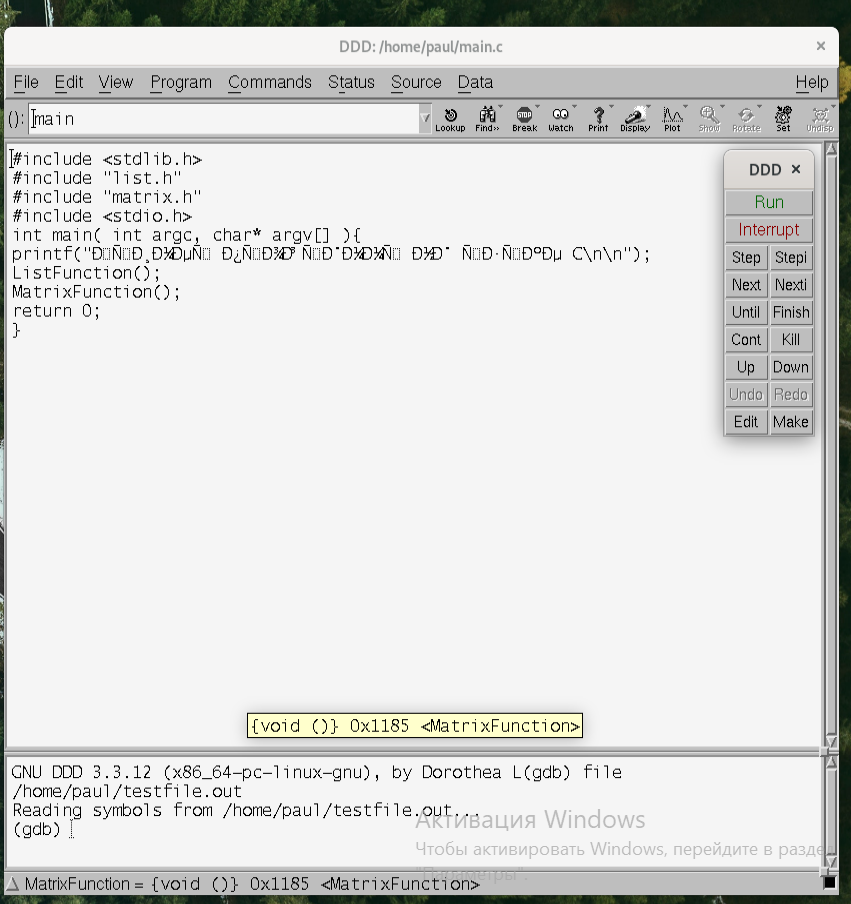
Продолжим выполнение программы:



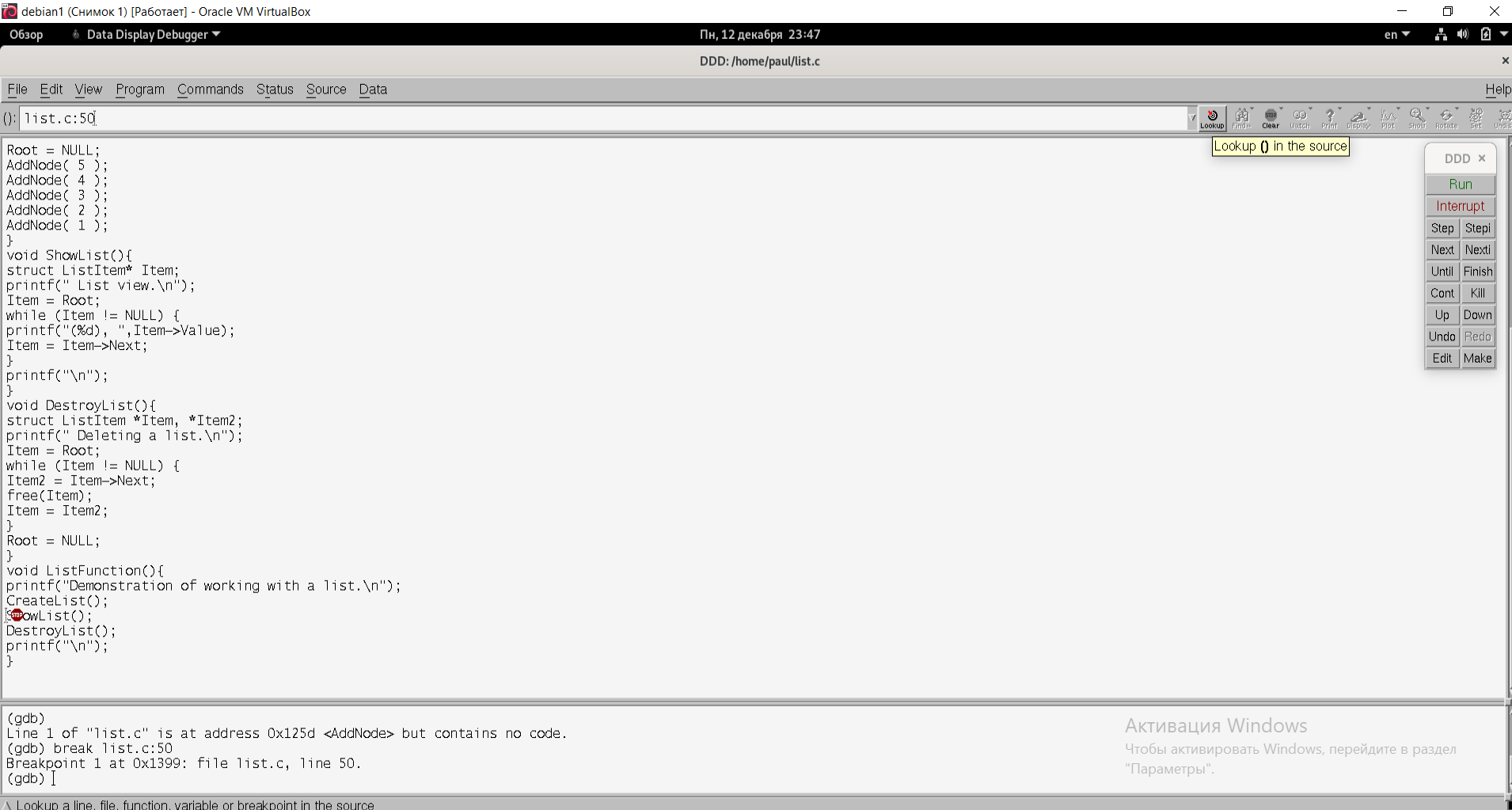
Как видно из последнего листинга, выполнение программы завершено. Выйдем из отладчика следующей командой: quit

**Задание 13. Далее выполним отладку с помощью отладчика Data Display Debugger. Данный отладчик реализует более привычную среду отладки с графическим интерфейсом. Также он обладает достаточно мощными средствами отображения отладочных данных. Запустим отладчик следующей командой:**

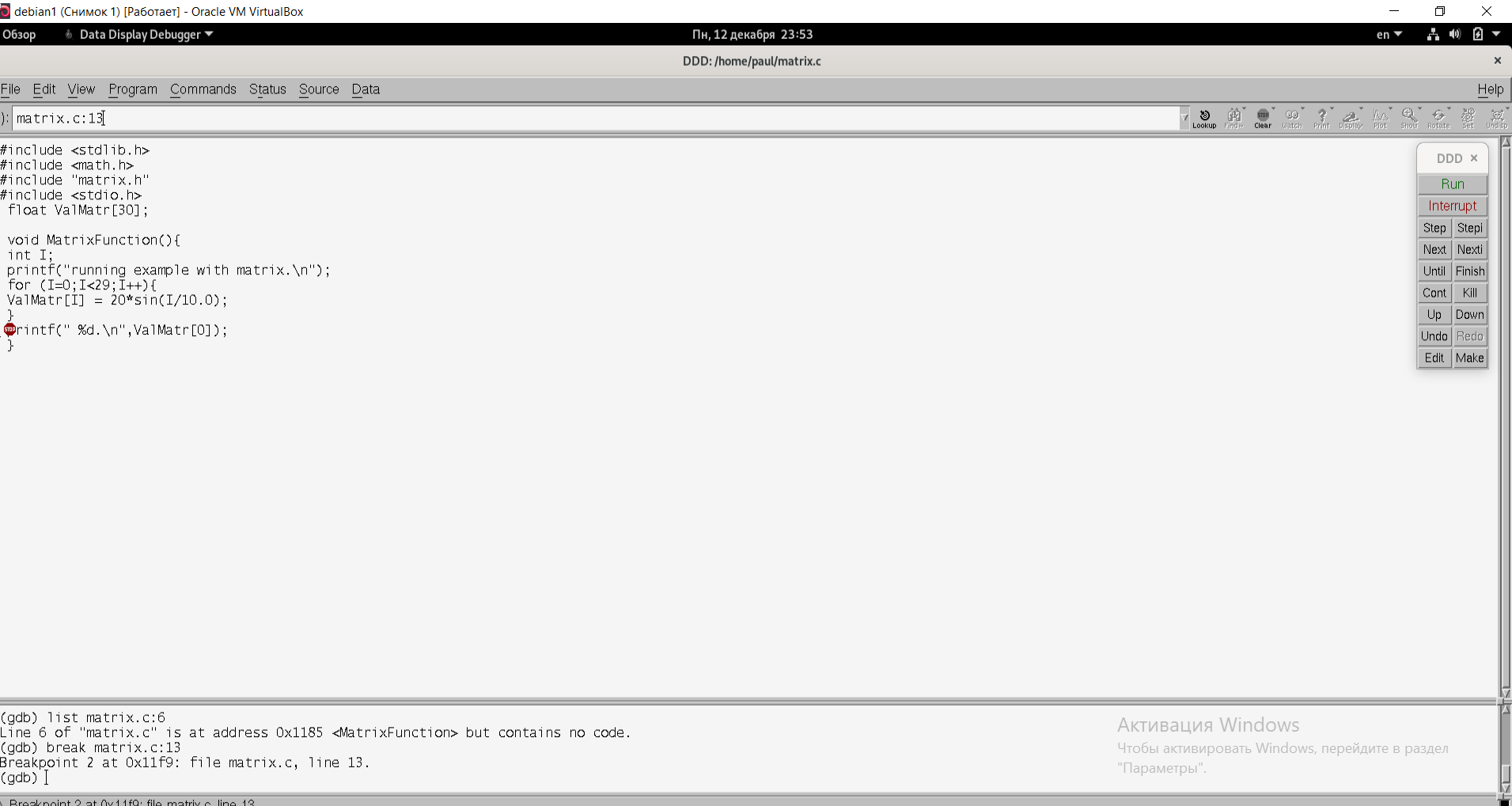
В результате на экране появилось окно отладчика. В меню «File» я выбрал пункт «Open Program». В появившемся диалоговом окне выбрал наш исполняемый файл testfile.out и нажал кнопку «Open». В результате окно отладчика приобрело вид:



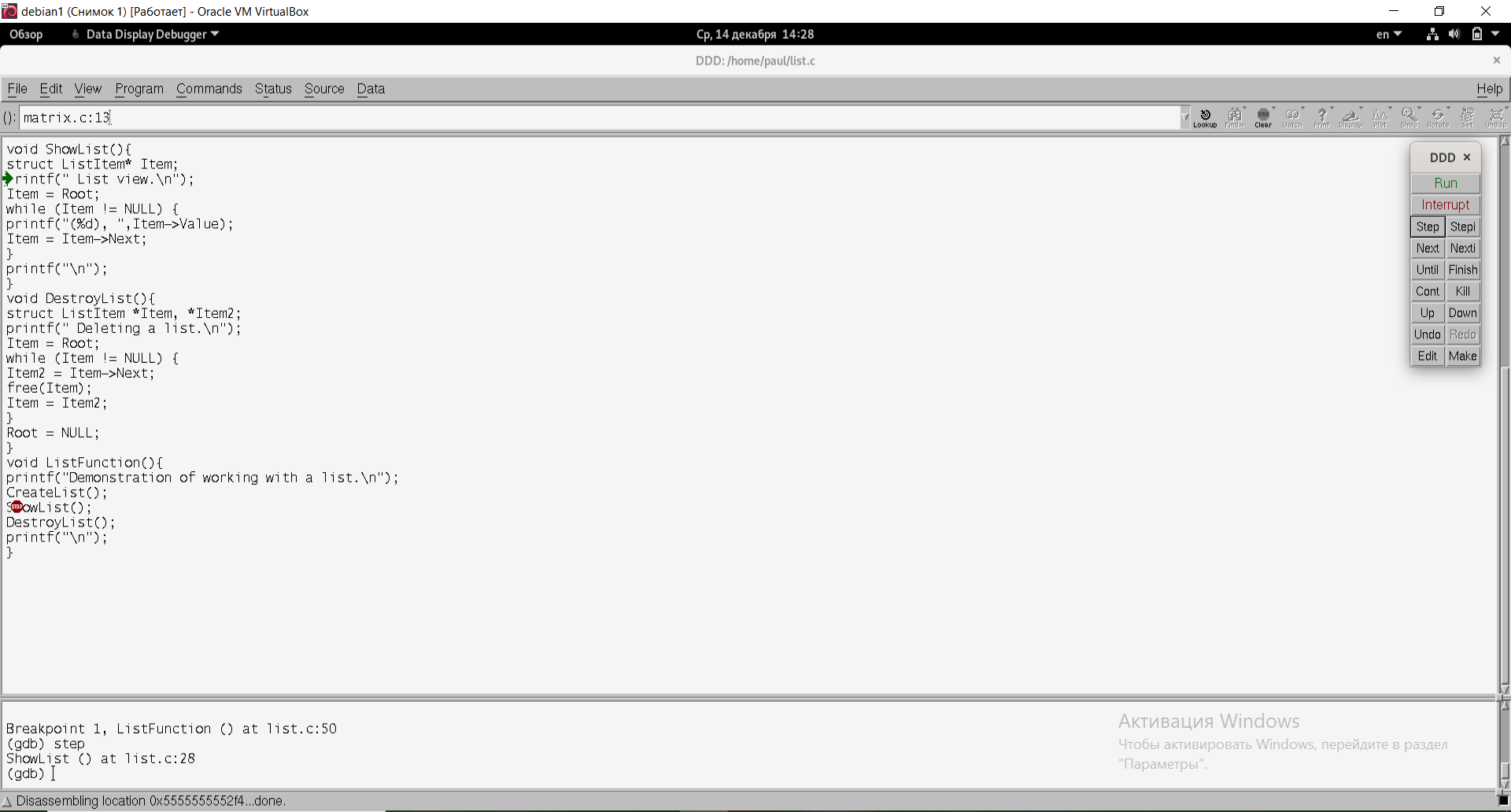
Открыл файл исходного кода list.c. Нашел в нем функцию ListFunction(), поместил курсор мыши в левый край строки, содержащей вызов функции ShowList(), и нажмал правую кнопку мыши. В появившемся контекстном меню выбрал пункт «Set Breakpoint». При этом левее данной строки кода появится пиктограмма, изображающая точку останова:



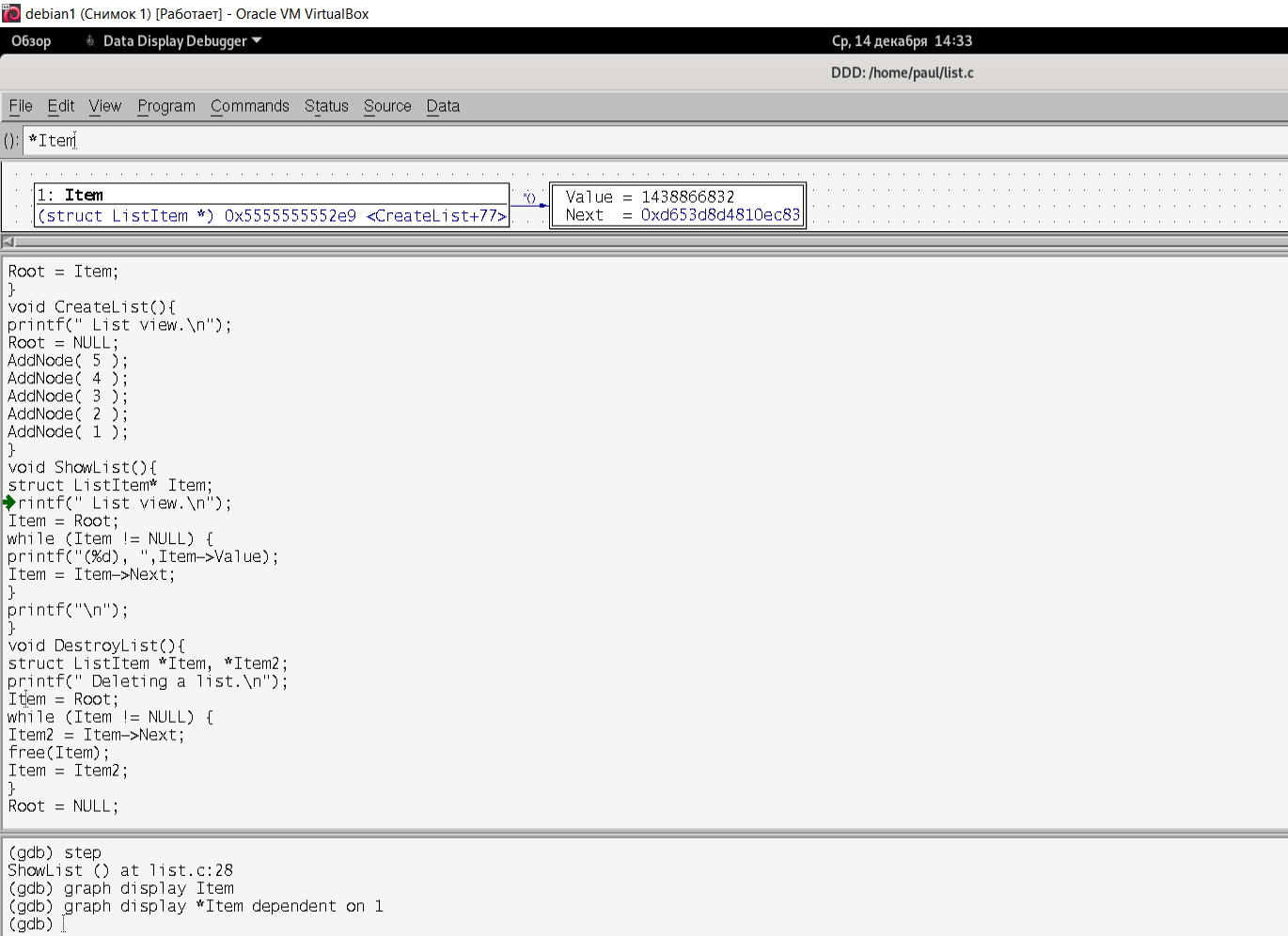
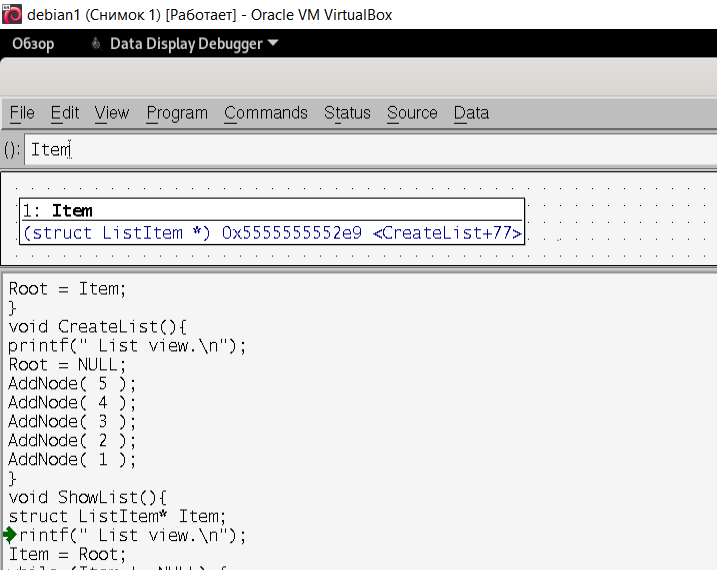
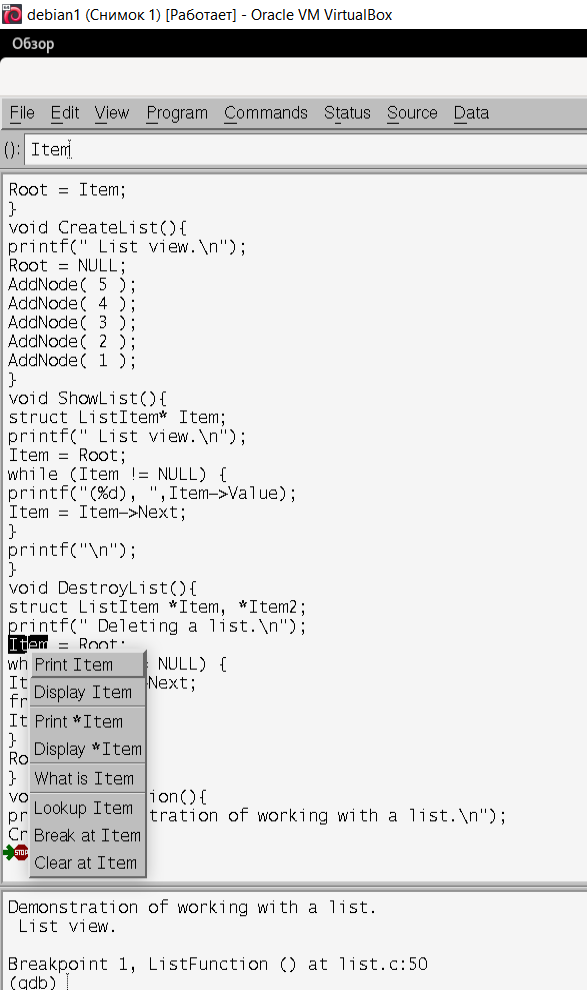
Откройте файл исходного кода matrix.c и поставьте точку останова в функции MatrixFunction() внутри цикла.

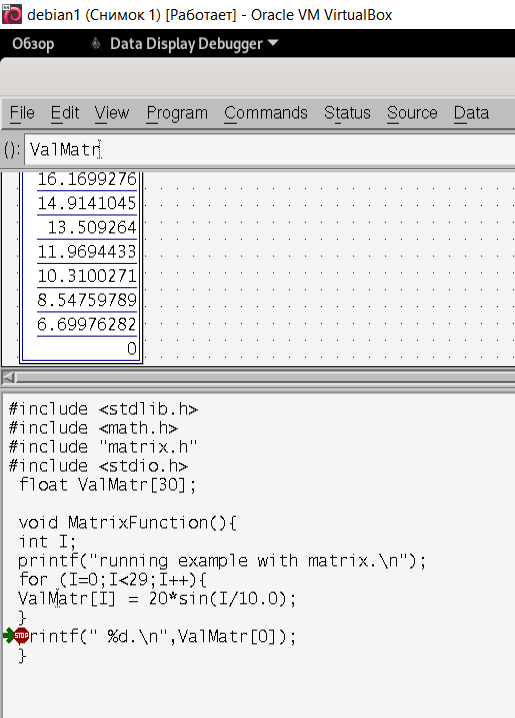


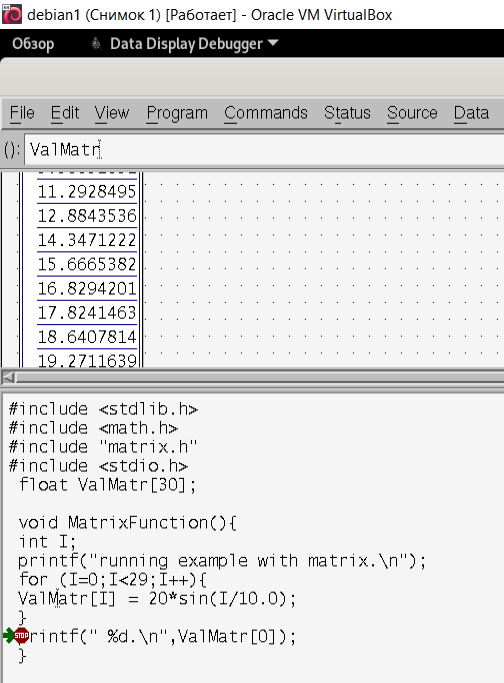
Запустите программу, нажав на кнопку «Run» в окне быстрого выбора команд. В результате выполнение программы будет остановлено при достижении первой точки останова. Нажимайте кнопку «Step» пока не войдете в функцию, и переменной Item не будет присвоено значение.

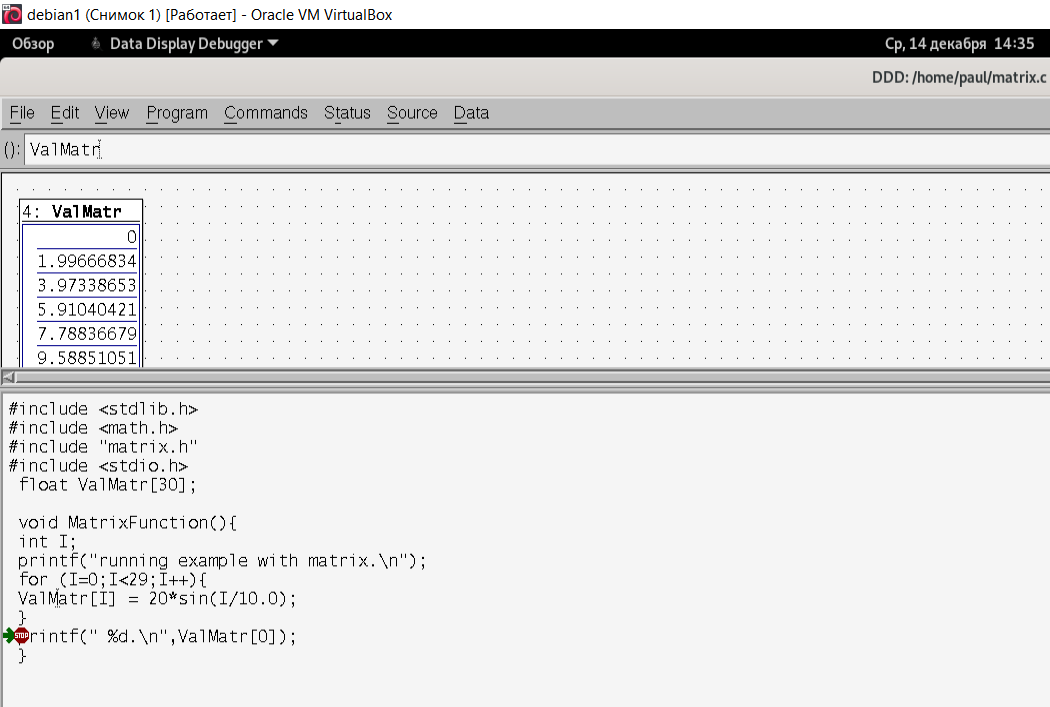


Продемонстрируем возможности отладчика по изображению списков. В области кода щелкните правой кнопкой мыши на названии переменной Item. В появившемся контекстном меню выберите «Display Item». При этом в области отображения данных появится прямоугольник, соответствующий данной переменной, в верхней части которого содержится имя переменной, а в нижней – значение адреса, на который ссылается указатель Item. В области отображения данных щелкните правой кнопкой мыши на данном значении адреса и в появившемся контекстном меню выберите пункт «Display \*()». В результате в области отображения данных появится изображение первого элемента списка, связанное стрелкой с изображением переменной Item

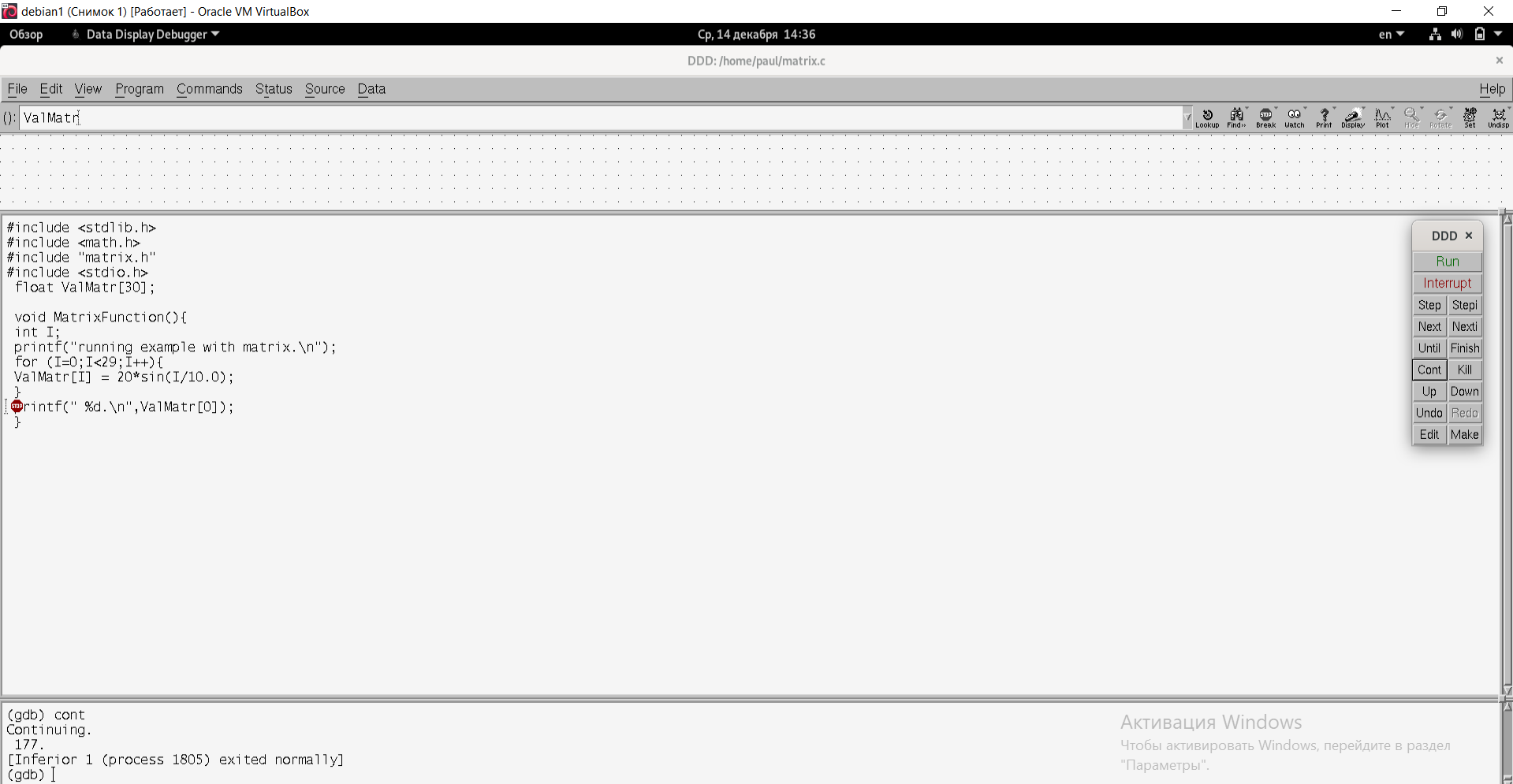


В окне быстрого выбора команд нажмите кнопку «Continue». Далее произойдет остановка выполнения программы на второй точке останова. Отобразите содержимое массива ValMatr в области отображения данных. Как видно, массив отображается в виде таблицы значений, что достаточно удобно**. Изображение массива в окне отладчика ddd: **

****

****

В окне быстрого выбора команд нажал кнопку «Continue». Работа программы завершилась. После этого я закрыл окно отладчика ddd.



**Контрольные вопросы:**

1. **Что собой представляет компилятор GCC? Какие основные аппаратные платформы и языки программирования им поддерживаются?**

**GCC** — это *набор компиляторов* разных языков программирования, созданный и поддерживаемый Фондом свободного программного обеспечения в рамках проекта GNU. Первоначально аббревиатура GCC расшифровывалась как GNU C Compiler (компилятор языка C проекта GNU). Позже GCC стали трактовать как **GNU Compiler Collection**(набор компиляторов проекта GNU).

Компилятор GCC может выполнять компиляцию программ, написанных на языках C, C ++, Objective C, Fortran, Java, Ada и Go. Также сторонними разработчиками поддерживаются дополнительные модули для компиляции программ на Pascal, Modula-2, Modula-3, Cobol, PL/1 и некоторых других языках. Компилятор GCC является стандартным компилятором в Linux, а так же в некоторых других Unix-подобных операционных системах. Помимо этого, существует **MinGW**— портированная версия GCC для Microsoft Windows, которая кроме самого компилятора содержит интерфейс для взаимодействия с Windows API.

1. **Что собой представляет процесс компиляции программы на языке C компилятором GCC? Каковы основные этапы данного процесса? Какие получаются промежуточные и конечные результаты?**

Компиляцией чаще всего называют процесс трансляции программы с языка программирования высокого уровня на машинно-ориентированный язык и последующую компоновку программы в готовый к использованию программный модуль. Это означает, что на входе у компилятора исходный текст программы, а на выходе — работающее приложение, написанное в машинных кодах. Этот процесс необходим, поскольку человеку гораздо проще писать и читать программы в виде, близком к человеческой речи. Машины же оперируют только понятиями команд и данных — и те и другие представляют собой числа в двоичной системе счисления.

1. **С помощью какой команды выполняется компиляция простейшей программы на языке C? Как выполняется компиляция программы, состоящей из нескольких файлов? Как выполняется компиляция программы, использующей статическую библиотеку? Как выполняется компиляция статической и динамической библиотеки?**

Выполнить компиляцию простейшей программы можно командой:

$ gcc имя\_файла

**1 способ**

gcc имя\_программы имя\_второй\_программы -o общее\_название   
$ ./общее\_название

Однако рассмотренный вариант в некотором смысле является не самым удачным. Дело в том, что при этом выполняется перекомпиляция всех программных файлов. Если же изменялись только некоторые из них, то удобнее применять другую методику компиляции – сначала получить для каждого файла объектный файл, затем выполнить компоновку объектных файлов.

**2 способ**

$ gcc имя\_программы -c   
$ gcc имя\_второй\_программы -c   
$ gcc имя\_программы имя\_второй\_программы -o общее\_название  
$ ./общее\_название

**Компиляция программы, использующей статические библиотеки**

Рассмотрим следующую программу (файл sin.c).

/\* sin.c \*/   
#include <stdio.h>   
#include <math.h>   
 int main (int argc, char\* argv[]){   
 float x, y;   
 printf("x=? : ");   
 scanf("%f", &x);   
 printf("x = %f\n", x);   
 y = sin(x);   
 printf("sin(x) = %f\n", y);   
 return 0;   
}

В ней используется функция**sin()**, описанная в заголовочном файле **math.h**. Попробуем скомпилировать программу обычным способом:

$ gcc sin.c   
/tmp/ccFEeYgH.o:In function `main':   
sin.c: (.text+0x48): undefined reference to `sin'   
collect2: ld returned 1 exit status

Как мы видим, компиляция завершилась неудачей — компоновщик (collect2) не смог найти местонахождение функции **sin()**. Данная функция содержится в файле статической библиотеки **libm.a**. Для успешной компиляции программы нужно указать компилятору на использование данного файла библиотеки. Для этого существует специальный параметр **–l<имя>** , где **<имя>** — имя файла библиотеки без трех первых букв lib и без расширения файла (по существующему соглашению все файлы статических библиотек начинаются с букв lib, поэтому их можно опустить). Таким образом, нам нужно указать в команде ключ **–lm**.

$ gcc sin.c -lm -o sin.out   
$ ./sin.out   
x=? : 3.1415926   
x = 3.141593   
sin(x) = 0.000000

**Статические и разделяемые (динамические) библиотеки**

Кроме исполняемых файлов с помощью компилятора GCC (и дополнительных утилит) мы можем еще скомпилировать библиотеки подпрограмм двух видов — *статические* библиотеки (\*.a) и *разделяемые* (динамические) библиотеки (\*.so).

Если мы хотим в своей программе использовать *статическую* библиотеку, то она нам понадобится уже на стадии компиляции. Мы уже рассматривали пример со статической библиотекой libm.a. Недостатком статических библиотек является то, что содержащийся в них код включается в каждый исполняемый файл, который использует библиотеку. Если одну и ту же библиотеку использует сразу несколько программ, то каждая из них содержит копию используемой библиотеки, что негативно отражается на потреблении памяти.

Если мы хотим использовать в программе *разделяемую* библиотеку, то она нам потребуется только на стадии выполнения программы. Важным достоинством разделяемых библиотек является то, что в памяти хранится только одна копия разделяемой библиотеки вне зависимости от количества работающих с ней программ.

**Компиляция статической библиотеки**

Статическая библиотека подпрограмм, по сути, представляет собой набор объектных файлов, которые помещаются в единый архив. Соответственно, нужно получить объектные файлы для включаемых в библиотеку подпрограмм, а потом сформировать из них статическую библиотеку с помощью специальной утилиты **ar**

В нашем случае при компиляции файлов исходного кода нужно указать параметр **–fpic**(от Position Independent Code) для того, чтобы был скомпилирован код, работающий независимо от физического расположения кода и данных в памяти.

$ gcc -fpic -c filelib1.c filelib2.c

Далее при окончательной компиляции нужно указать параметр **–shared**, который определяет, что будет создана разделяемая библиотека.

$ gcc -shared filelib1.o filelib2.o -o libsharedexample.so   
$ ls   
filelib1.c filelib1.o filelib2.c filelib2.o libsharedexample.so

Как видно из листинга, разделяемая библиотека **libsharedexample.so** успешно создана.

1. **Как при компиляции задать используемый стандарт языка C? Как включить в исполняемый файл отладочную информацию? Какие существуют режимы оптимизации кода в компиляторе GCC? Какие существуют параметры выдачи предупреждений компилятора?**

**Указание используемого стандарта языка C**

Параметр, касающийся компиляции программ на языке C**–std**, определяющий, в соответствии с каким из стандартов языка C выполнять компиляцию. Формат параметра: **-std=стандарт**, где в качестве стандарта может стоять одна из следующих констант: c89 (стандарт ISO C89), с99 (стандарт ISO C99), gnu89 (ISO C89 с некоторыми расширениями GNU), c11 (стандарт ISO C11).

**Включение отладочной информации**

Если вы собираетесь отлаживать программу, используя какой-либо отладчик, то в исполняемый файл необходимо включить *отладочную информацию*. Для этого в команде надо указать ключ **–g**.

**Параметры оптимизации кода**

При генерации исполняемого файла компилятор может тем или иным способом попытаться выполнить оптимизацию кода программы. Целями такой оптимизации могут быть:

* ускорение работы программы (за счет уменьшения числа выполняемых операторов и их перегруппировки);
* уменьшение объема используемой оперативной памяти;
* уменьшение размера исполняемого файла в байтах.

Если вы хотите выполнить генерацию исполняемого файла без оптимизации, например, для того чтобы получить ожидаемые результаты во время отладки, нужно указать ключ –O0. Впрочем, данный ключ можно и не указывать, он действует по умолчанию.

Для того чтобы выполнить оптимизацию, нужно указать один из ключей: **-O1, -O2, -O3, -Os**.

Ключ **–O1** включает базовый набор методов оптимизации (оптимизация уровня 1).

Ключ **–O2** включает все методы оптимизации уровня 1, а так же ряд дополнительных методов.

Ключ **–O3** включает все методы оптимизации уровня 2, а так же некоторые другие методы.

Существует также еще один флаг **–Os**, позволяющий уменьшить объем исполняемого файла. Данный флаг активизирует все методы 2 уровня оптимизации (как при использовании флага **–O2**), добавляя к ним ряд методов, направленных специально на уменьшение объема исполняемого файла.

При переходе от первого уровня оптимизации ко второму и от второго к третьему время компиляции растет, а эффективность оптимизации увеличивается.

**Параметры выдачи предупреждений**

Если вы попытаетесь выполнить компиляцию программы, которая содержит ошибки, то компилятор выдаст вам список ошибок. Даже если формально в программе нет ошибок, компилятор может выдавать ***предупреждения* (*warnings***), сообщающие о том, что в программе применяются какие-либо не рекомендуемые средства или методы программирования, например, используются функции, считающиеся небезопасными.

**Программист может указывать компилятору, какие из предупреждений следует выдавать программисту.** Обычно, ключ, управляющий выдачей предупреждений некоторого типа, имеет формат **–Wпредупреждение**.  Достаточно часто используется ключ **–Wall**, указывающий на то, что должно выдаваться большинство существующих предупреждений. Еще больше предупреждений выдается при указании ключа **-Wextra**.

1. **Из каких структурных частей состоит компилятор GCC? Какие в нем существуют промежуточные формы представления программы?**

**Структура компилятора: препроцессор, компилятор, ассемблер, компоновщик.**

Компиляция программы на языке C состоит из следующих этапов:

* обработка препроцессором;
* собственно компиляция;
* ассемблирование;
* компоновка.

В процессе компиляции создаются еще файлы трех промежуточных типов.

**Типы файлов, с которыми работает GCC при компиляции программ на языке C**

|  |  |
| --- | --- |
| Расширение файла | Описание |
| .a | Статическая объектная библиотека (архив) |
| .c | Исходный код на языке C, подлежащий обработке препроцессором |
| .h | Исходный код заголовочного файла на языке C |
| .i | Исходный код на языке C, не подлежащий обработке препроцессором. Файлы данного типа являются промежуточным результатом компиляции |
| .o | Объектный файл в формате, поддерживаемом компоновщиком. Файлы данного типа являются промежуточным результатом компиляции |
| .s | Ассемблерный код. Файлы данного типа являются промежуточным результатом компиляции |
| .so | Разделяемая объектная библиотека |

1. **Что собой представляет отладчик GDB? Перечислите основные возможности и основные встроенные команды данного отладчика.**

На сегодняшний день стандартом де-факто среди отладчиков в Unix-подобных операционных системах является отладчик **GDB (GNU Debuger),** разработанный в рамках проекта GNU. Данный отладчик позволяет отлаживать программы, разработанные на языках программирования C, C++, Go, Pascal, Basic, Ada, Objective-C и других языках. Он широко используется как сам по себе, так и вместе с другими отладчиками и средами разработки, которые, по сути, являются высокоуровневой надстройкой над GDB.

**Отладчик GDB является одним из самых развитых отладчиков. Как и большинство других отладчиков, он позволяет:**

* запустить программу, указав все необходимые параметры запуска;
* остановить выполнение программы при указанных условиях;
* изучить состояние программы в момент ее останова;
* изменять значения переменных и регистров во время выполнения программы, что позволяет экспериментировать, с целью выявления и устранения ошибок в программе.

Отладчик GDB позволяет, как запустить программу из отладчика, так и подключиться к уже запущенному процессу.

***Основные команды отладчика GDB***

|  |  |  |
| --- | --- | --- |
| Имя и формат команды | Сокр. | Описание |
| Команды общего назначения | | |
| help |  | Вывести справку о работе с отладчиком GDB |
| file имя\_файла |  | Загрузить программу для работы под отладчиком (если программа не была указана как параметр при запуске GDB) |
| list номер\_строки | l | Вывести строку программы с заданным номером и 10 ближайших строк |
| quit | q | Выход из отладчика |
| Выполнение и трассировка программы | | |
| run | r | Запустить программу на исполнение, в команде можно указывать параметры командной строки и перенаправления ввода/вывода |
| start |  | Начать пошаговое выполнение |
| next | n | Выполнить следующую строку с обходом функций. В команде может быть указано число выполняемых операторов |
| step | s | Выполнить следующую строку с входом в функции. В команде может быть указано число выполняемых операторов |
| return |  | Выйти из текущей функции, пропустив оставшуюся ее часть |
| continue | c | Продолжить выполнение программы |
| kill | k | Прекратить выполнение отлаживаемой программы |
| Работа с точками останова | | |
| break номер\_строки | b | Установить точку останова в строке с заданным номером. В качестве аргумента могут выступать также адрес или имя функции |
| watch имя\_переменной | wa | Установить точку останова при изменении значения заданной переменной |
| info breakpoints |  | Отобразить информацию обо всех точках останова |
| disable breakpoint номер\_точки\_останова |  | Временно отключить точку останова с заданным номером (номер отображается в команде info breakpoints) |
| enable breakpoint номер\_точки\_останова |  | Включить точку останова с заданным номером (номер отображается в команде info breakpoints) |
| delete breakpoint номер\_точки\_останова |  | Удалить точку останова с заданным номером (номер отображается в команде info breakpoints) |
| Работа с переменными | | |
| print имя\_переменной |  | Показать значение заданной переменной |
| display выражение |  | Добавить выражение или переменную в список вывода. После выполнения данной команды значение выражения или переменной будет выводиться после каждого останова или выполнения шага трассировки |
| set var имя\_переменной=значение |  | Изменить значение указанной переменной |

1. **Что собой представляет отладчик DDD? Каким образом он связан с отладчиком GDB? Каковы основные достоинства отладчика DDD?**

Как видно из названия, отладчик **DDD(Data Display Debugger)** специализируется на удобном отображении текущих данных программы во время отладки. Когда выработаете с отладчиком, ваша работа заключается, прежде всего, в том, что вы пытаетесь понять текущее состояние программы и определить, в каком месте программы ее состояние становится ошибочным. Состояние программы – это в первую очередь совокупность текущих значений ее переменных, которые требуется одновременно держать в голове. Поэтому отладчик **DDD**, позволяющий упростить восприятие данных, является достаточно полезным инструментом.

**Основные достоинства отладчика DDD**

**Во-первых**, в отладчике DDD реализован традиционный интерфейс отладчика, такой как в интегрированных средах разработки. Окно отладчика разделено на три области. В верхней части окна располагается область отображения данных, в средней — область исходного кода программы, а в нижней части окна выводится консоль отладчика **GDB**.

**Во-вторых**, отладчик **DDD** позволяет отображать графически динамические структуры данных. Более того, отладчик **DDD** может автоматически обновлять изображение при пошаговой отладке. Поиск ошибок в работе с указателями является достаточно сложной задачей, и рассматриваемая функция отладчика является весьма полезной.

**В-третьих**, отладчик **DDD** позволяет отображать данные, хранящиеся в массивах, в виде графиков.