Ubuntu 9.10 programming environment

Making first steps

David Sariel (davidsa@openu.ac.il)

June 2010

Table of contents

Preface

This short manual was issued to introduce students studying the Operating Systems course to the Linux operating system.

The main goal was to spare the venture of Linux installation and system configuration because the vast majority of our students see Linux system in their very first time. Driven by this reason the way proposed is to run Linux as a guest operating system hosted by some version of Windows operating system. Student are required to install VMware Player that was developed by *VMware inc*. Then the Linux system could be launched in the window of the host operating system.

Operating systems course supplies Linux virtual machine configured in such a way that allows it to borrow several vital configurations and hardware recourses from the host operating system. Linux running as a guest operating system will receive access to CD-ROM, audio devices and internet connection from the host operating system. To store data the Linux virtual machine is equipped with 20GB virtual hard disk that is not shared with the host operating system.

I would like to conclude with expression of especial gratitude to Ehud Lamm for thorough review, useful advises and prodding for better prose and to Dr. Jacky Hertz both for reviewing the entire brochure and for his efforts of interweaving Linux into the CS courses. Also, I would like to mention that Karl Schock's (www.elele.de/knoppix/docs/tutorial) tutorial was of great help together with valuable information retrieved from pages of Ubuntu project. Section about libraries was taken from lecture summaries of Alex Kremer and revised. Section debugging was taken from the homepage of the DDD debugger.

Part One

Launching Virtual Machine

Introduction

Operating systems course supplies the Linux virtual machine based on the Ubuntu operating system. Ubuntu is an open source project comprising a collection of GNU/Linux software. The system suits our main goal because of automatic hardware detection and because if comes with vast amount of preinstalled applications. Following is a partial list of what you got:

- ✓ Linux Kernel 2.6
- ✓ GNOME Desktop manager
- ✓ Open Office
- ✓ Eclipse
- ✓ Emacs
- ✓ DDD
- ✓ Lots of games
- ✓ Utilities for data recovery
- ✓ Internet browsers

Ubuntu is based on Debian Linux. Source files for any of the installed packages may be downloaded from the Ubuntu Linux mirrors.[1]

[1] sudo apt-get build-dep $package
where package=whatever package you want.

then do
sudo apt-get source $package
to get the source for that package.

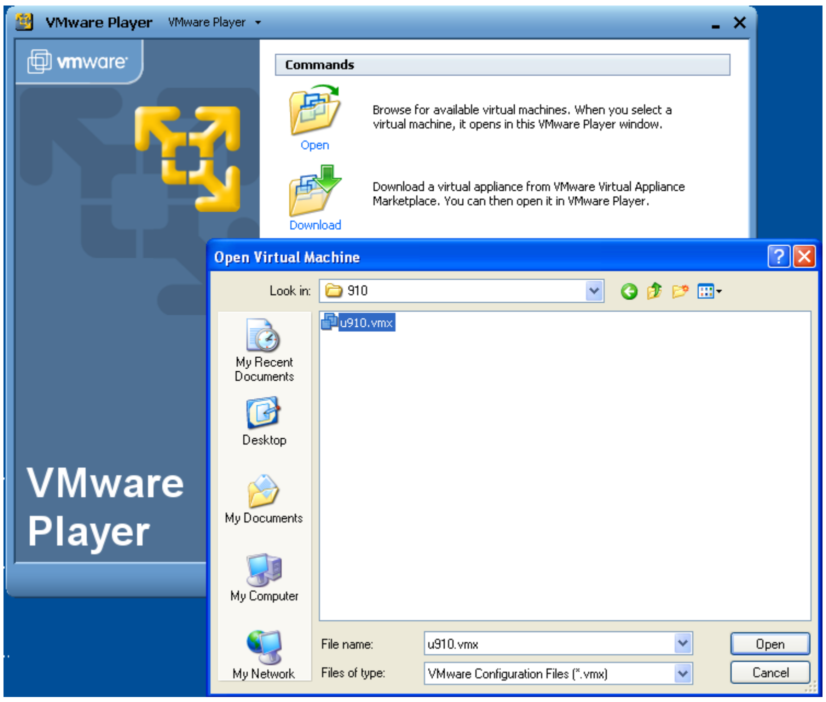
Hardware requirements

To run the Ubuntu virtual machine, you need:

- ✓ Intel-compatible CPU
- ✓ 768 MB of RAM (1.5 GB is preferred)
- ✓ Standard SVGA-compatible graphic cars
- ✓ Mouse: serial or PS2 or USB
- ✓ Space on the HD (min 5 GB – max 43 Gb)

Booting Ubuntu virtual machine

In order to boot Ubuntu virtual machine it should be first copied to your hard disk. Copy the entire content of the DVD labeled "Ubuntu – virtual machine" supplied by the Operating course to a partition with at least 7 GB free space. Turn those files from being 'read only' to be 'read/write'. Install the VMware Player (from the www.vmware.com) and run it. You will be prompted to specify the location of your Ubuntu virtual machine:


Browse for available virtual machines. When you select a virtual machine, it opens in this VMware Player window.

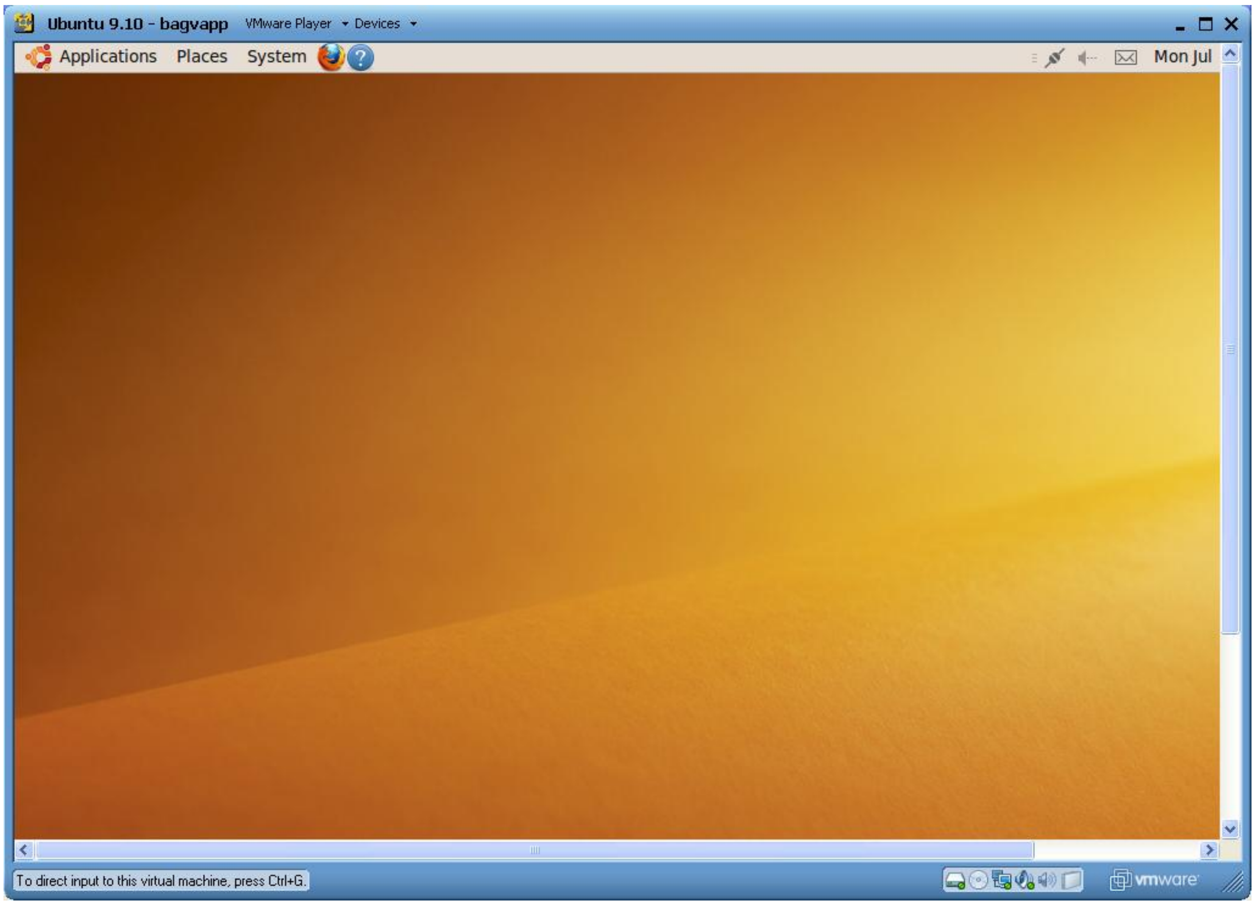
Download a virtual appliance from VMware Virtual Appliance Marketplace. You can then open it in VMware Player.

Look in: 910

File name: u910.vmx

Files of type: VMware Configuration Files (*.vmx)

A short while after the following GNOME graphical user interface appears:



To direct input to this virtual machine, press Ctrl+G.

In order to continue working with Ubuntu your virtual machine needs to **get a focus**. This could be done either by:

- Maximizing the VMware Player window and running Ubuntu in the full screen mode

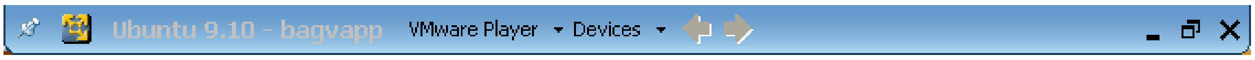
or by

- Clicking the mouse inside the VMplayer.

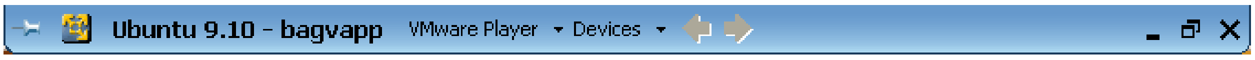
NOTE: Hitting “ALT+CTRL” combination returns back to the Windows system.

RECOMMENDATION: Full screen mode gives a feeling of working with a real machine installation.

While working is the full mode you may either pin in the VMWare pull up menu:



or to allow him pulling back and forth:

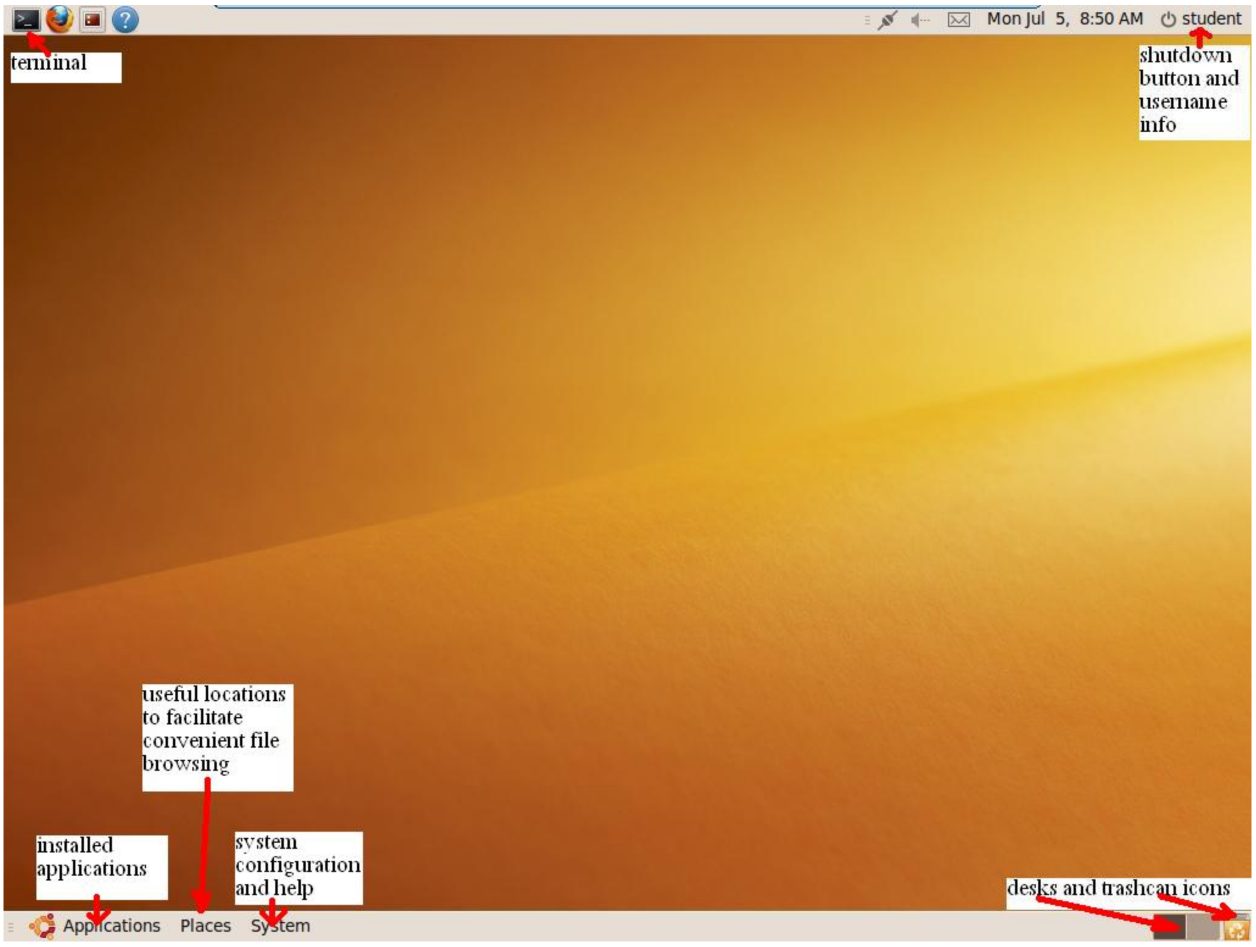


The GNOME desktop manager

GNOME desktop manager is an appealing and easy to configure graphical environment. The goals of the GNOME are to lessen the Linux/UNIX learning curve for many users, and to present a friendly environment for everyone.

GNOME is not the only graphical environment for Linux workstations. But every environment, including GNOME, supports the so-called X-protocol, which is necessary to communicate with X11 server. X11 is a distributed windowing system that allows you to open many windows on the screen and perform several different tasks in parallel: login to more than one machine, run applications on remote machines, etc. It allows the user to have a lot of control over the display: choose fonts, colors, size and placement of windows, etc. A program communicating with X11 system is called a client. The GNOME graphical environment itself is a special client called a window manager.

When you first start GNOME, among the first items you'll see are:


Mon Jul 5, 8:50 AM
student
terminal
shutdown button and username info
useful locations to facilitate convenient file browsing
installed applications
system configuration and help
desks and trashcan icons
Applications
Places
System

The Desktop -- The main workspace of your environment; the space on which you place windows for running applications, icons for starting programs, folders for programs or documents.

The Desks and Trash icons -- These are links (or shortcuts) which allow you easy access the trashcan content and multiple desks. You can spread your work throughout multiple desks, rather than crowd one desktop area with large number of applications.

The shutdown button -- Provides shutdown, hibernation, standby, logout etc. options.

Terminal – Unix operating systems were originally designed as text only systems controlled by keyboard commands called **command line interpreter (CLI). The X-window** system has been added since then to provide a graphical interface. However, the underlying CLI is still there and is frequently the easiest, fastest and most powerful way to perform many tasks. Terminal or Xterm (the X-Windows terminal emulation program) includes, among many other features, the ability to use multiple terminal shells in a single window, making for a less cluttered desktop.

Applications – This menu provides organized access to the applications installed on your virtual desktop.

Places – The menu of shortcuts to frequently used locations.

System – This menu facilitates system configuration and getting help.

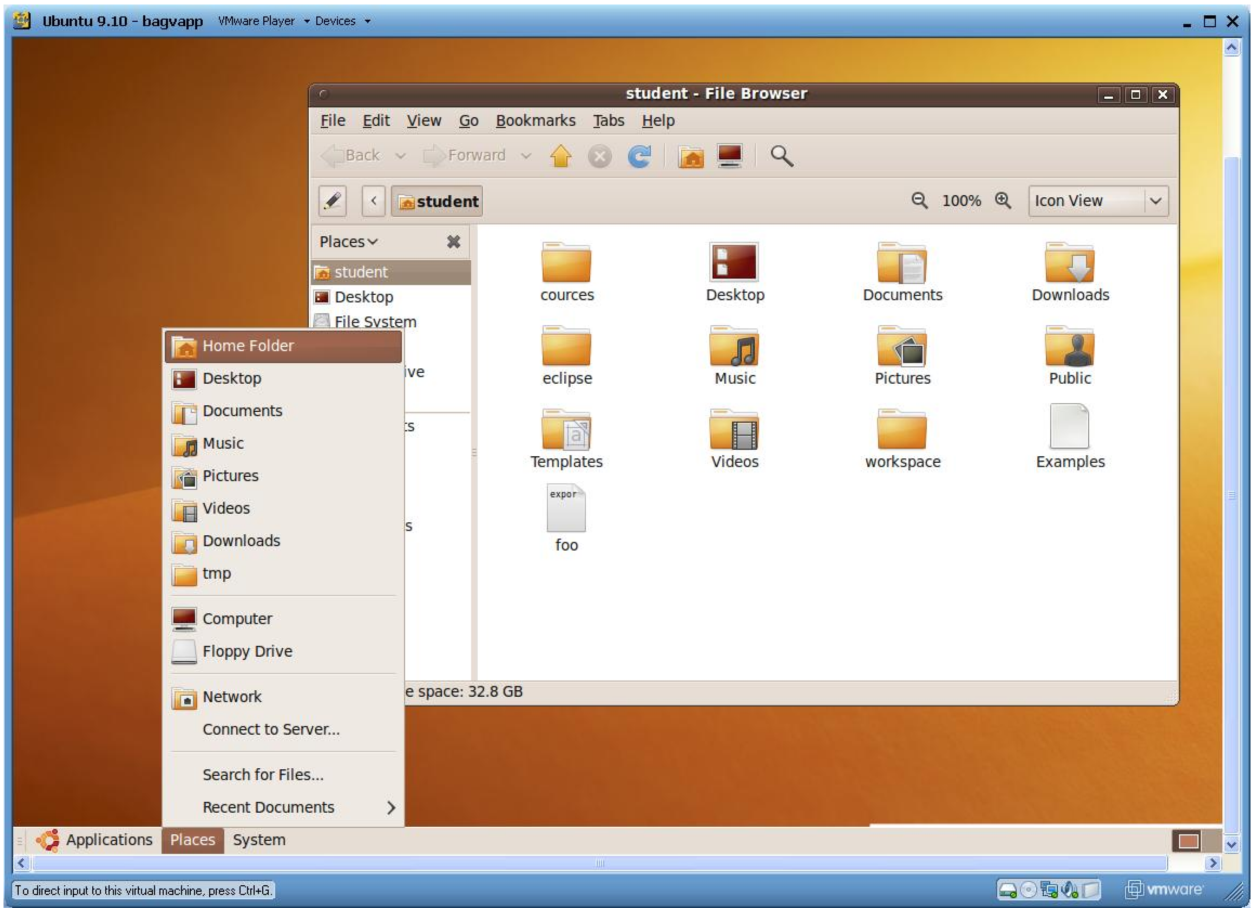
Configuring VMware Player

The VMware Player needs almost no configuration. But if you have enough RAM you can find it useful to enlarge the amount of RAM used by your virtual machine. Graphical environment’s stability will definitely benefit from the enlarged RAM amount. I have 1.5 GB installed on my machine and running Ubuntu virtual machine with 512 MB preallocated RAM, which leaves 1 GB for the host operating system (Windows).

Where to save files

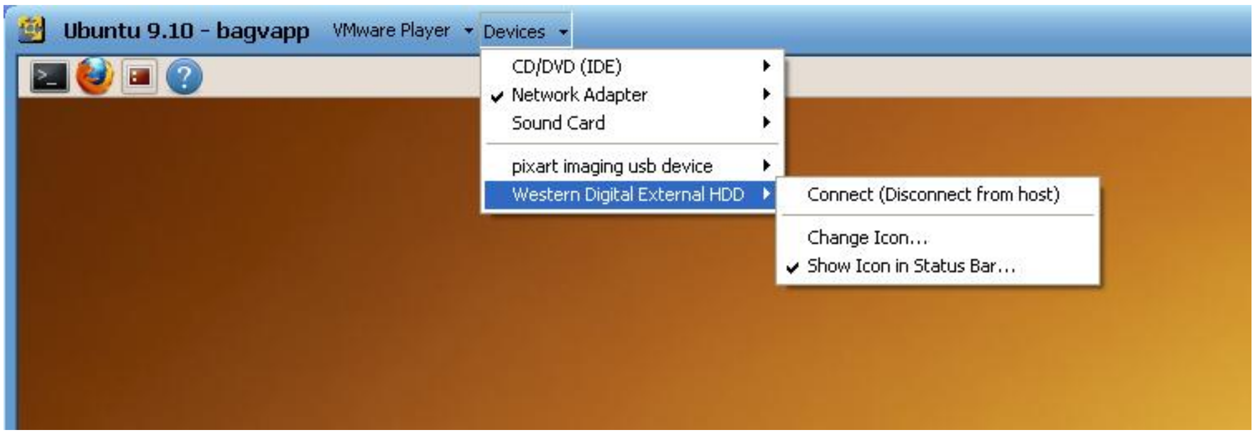
Virtual disk

The Ubuntu virtual machine supplied by the Operating systems course comes with virtual SCSY disk with maximal capacity of 38 GB. Through the Places shortcut you may open your personal folder (homedir) and save your files therein. For example, the folder cources/ already contains The folder named os/ where you can store Operating Sytems course related material.



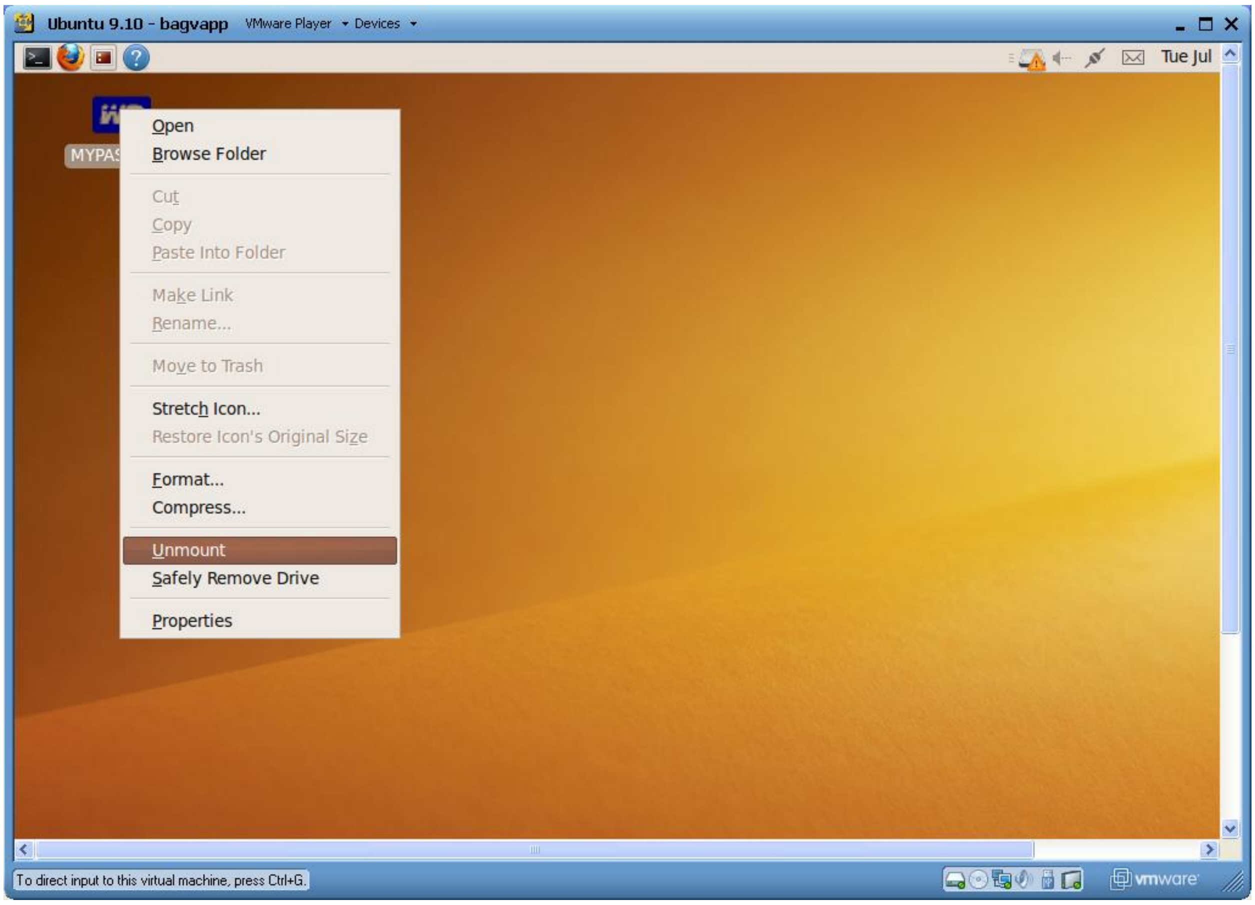
External USB devices

USB disks or DiskOnKey devices may be connected to the Ubuntu virtual machine. Ubuntu will detect them only if they were connected to the virtual machine:


Ubuntu 9.10 - bagvapp
VMware Player
Devices
CD/DVD (IDE)
✓ Network Adapter
Sound Card
pixart imaging usb device
Western Digital External HDD
Connect (Disconnect from host)
Change Icon...
✓ Show Icon in Status Bar...

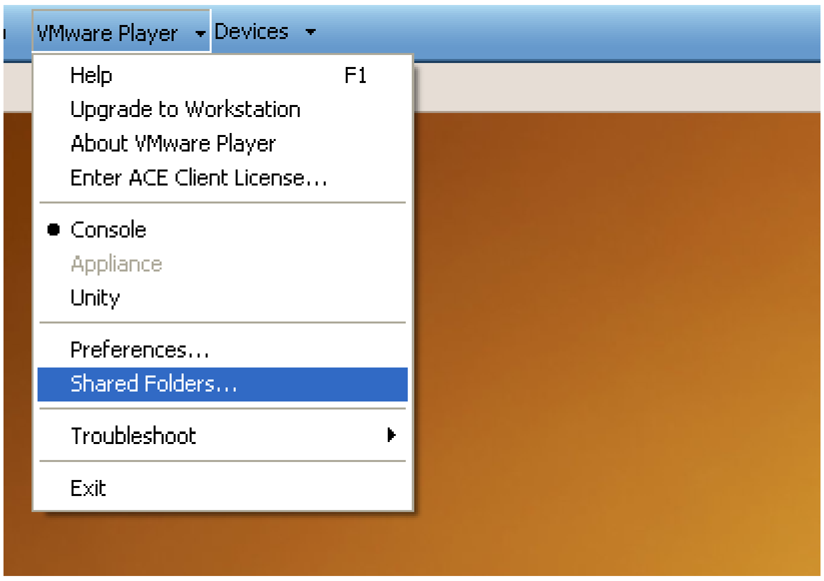
When Ubuntu detects a new USB device an appropriate icon appears on the Desktop. Since USB devices are interpreted via the SCSI interface, connected USB device will appear named sdbX, where X is a number ranged 1-16.

WARNING: Always perform "unmount" before you unplug the USB device (or at least run the filesystem synchronization command *sync* from the terminal that flushes filesystem buffers). Click the right mouse button on the disk icon and choose "Unmount". Make sure to quit ALL the tasks that use the USB device to be unmounted, otherwise "unmounting" fails. To connect the USB device back to the host operating system, disconnect the device from the VMWare player (via the Devices menu).

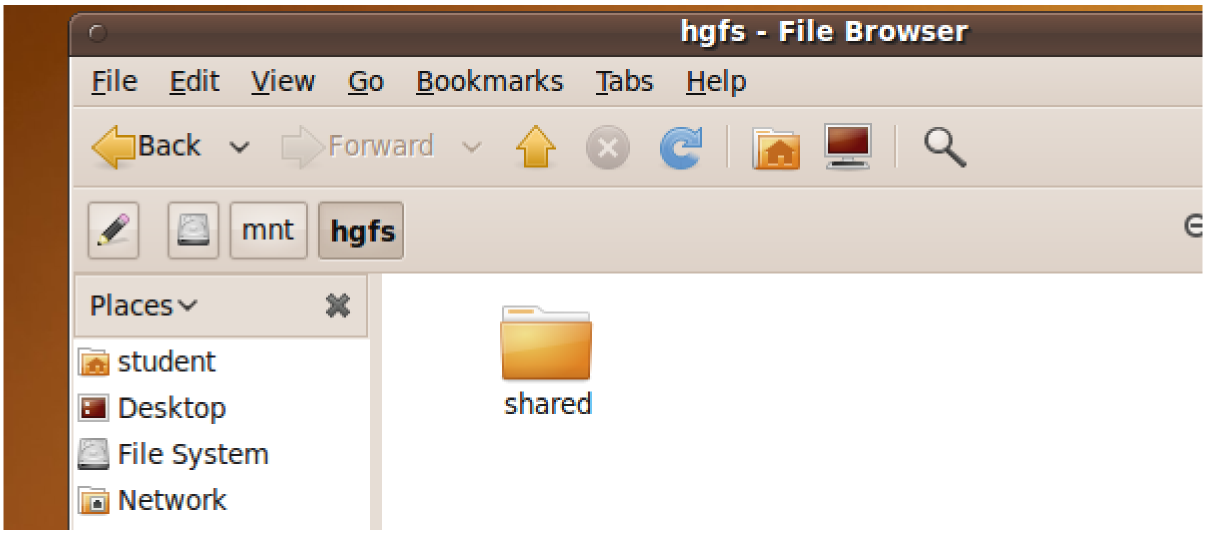


Shared directory

You can establish sharing between the guest (Ubuntu) and host (Windows) operating systems. Create a folder that will be shared between the systems (e.g. C:\shared). Then, turn VMPlayer to recognized it:



When finished, shared folder will be available at /mnt/hgfs/shared:



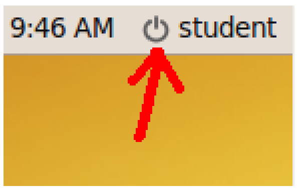
Floppy disk

If you have 3.5” floppy drive, you can save files on diskettes using `mtools`. `mtools` is a collection of DOS-like commands `mdir` `mcopy`, `mdel` etc, designed especially to work with diskettes formatted FAT-12 from the command line. (FAT-12 is a format recognized by Windows operating systems). Following is a table of examples:

Goal	Command
mformat a:	Format the floppy that is in drive A:
mdir a:	List the files on the floppy in drive A:
mcopy file1 a:	Copy the file “file1” from the current directory to the floppy in drive A:
mcopy a:file1 .	Copy the file “file1” from the floppy in drive A: to the current working directory on the Linux system. “.” stands for the current working directory.

Shutting the system down

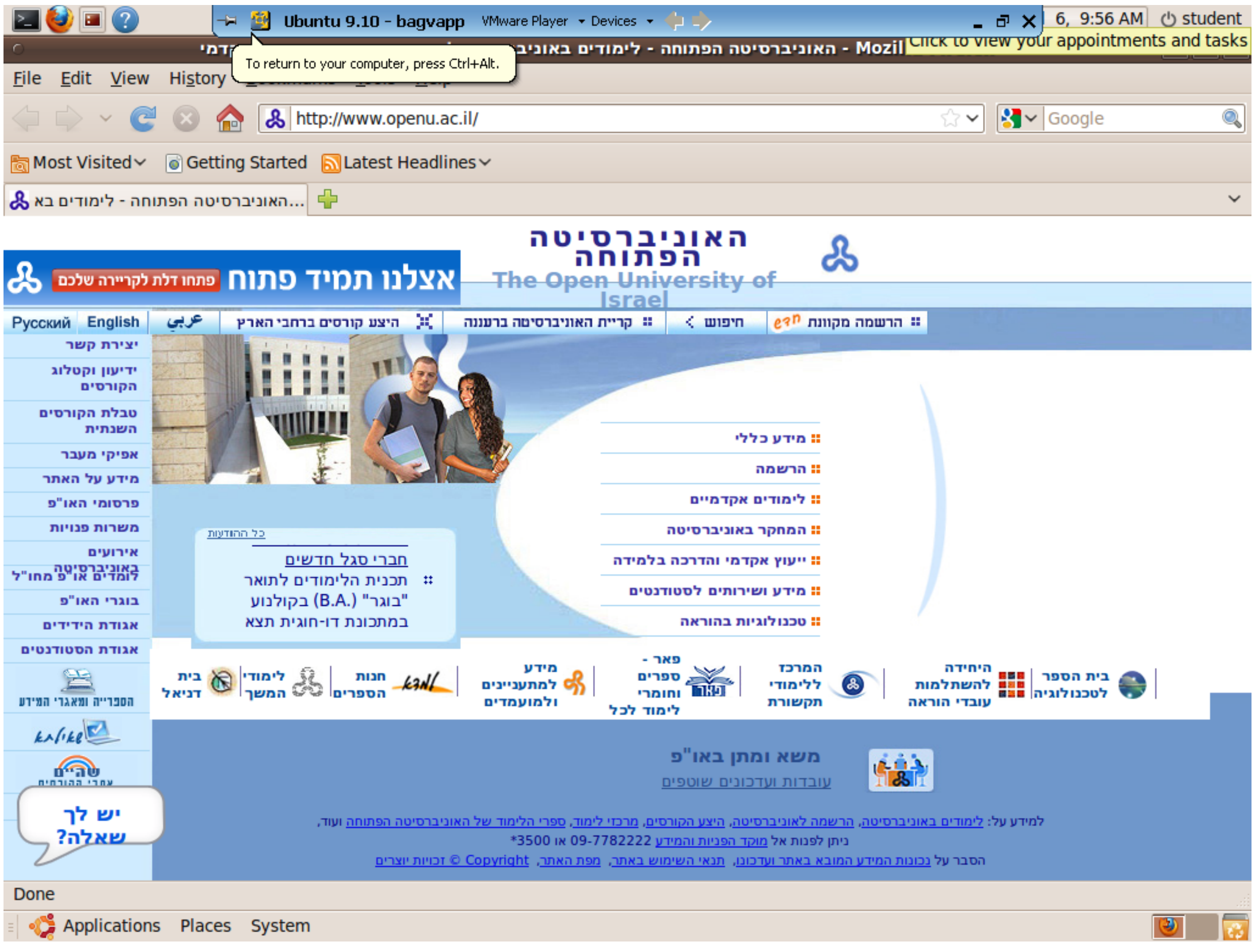
To terminate the session and shut down the virtual machine, just click on the shutdown button.



WARNING: Linux systems are different from Windows systems in the way they operate their hard disks. Saving file in Windows system causes it to be immediately written to the hard disk. Linux systems often defer physical writing in order to optimize disk arm motions. Therefore, in order to avoid loosing data don’t power off VMware Player from Windows.

Connecting the Internet

The virtual machine was configured to use the Internet connection of the host operating system. Establish the Internet connection from Windows and switch to the virtual machine. Launch Firefox Internet browser and start browsing the Internet:



Part 2

Programming environment

How to get some help?

Linux has a very extensive online documentation system documenting every available command and many library routines. To get manual page run from the terminal "man" command followed by the subject of your interest.

For example, launch a terminal and type "man ls" in the command line. Comprehensive help on the "ls" command (which is similar to the "dir" command in MSDOS) will be displayed.

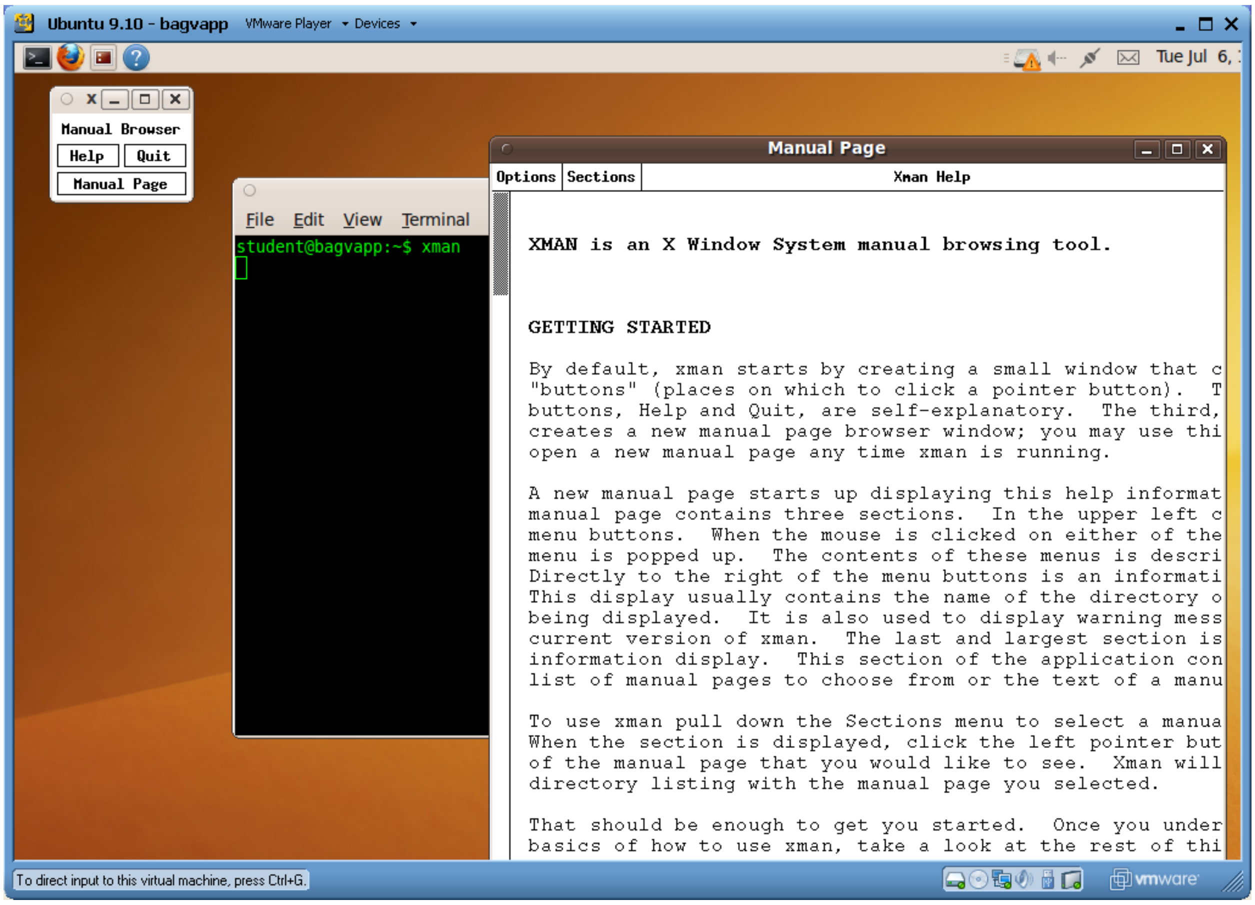
The manual (man pages) is organized in sections. Section 1 describes general user commands and sections 2 and 3 are a Linux programmer manual. The first line in the output of the "man" command displays the section number (see the top left corner of the screen). Sections 2 and 3 will be the most widely used preparing your programming assignments.

Sometimes, a subject of your interest may have similar syntax with other command, function etc. For example, typing "man read" displays the information related to the Tcl built-in command in the "Tcl built-in commands" section, while typing "man 2 read" supplies the library function description from the Linux programmer's manual (man-pages, section 2).

If the exact syntax is unknown, use the flag `-k`. For example, "man –k read" displays a list of subjects that have a string "read" in their short description of the manual pages database. Then one can choose which commands to read about.

You can alternatively use a graphical version of "man" invoked by the "xman" command. Click on the button named "Manual pages" and then select "Section" from the menu.

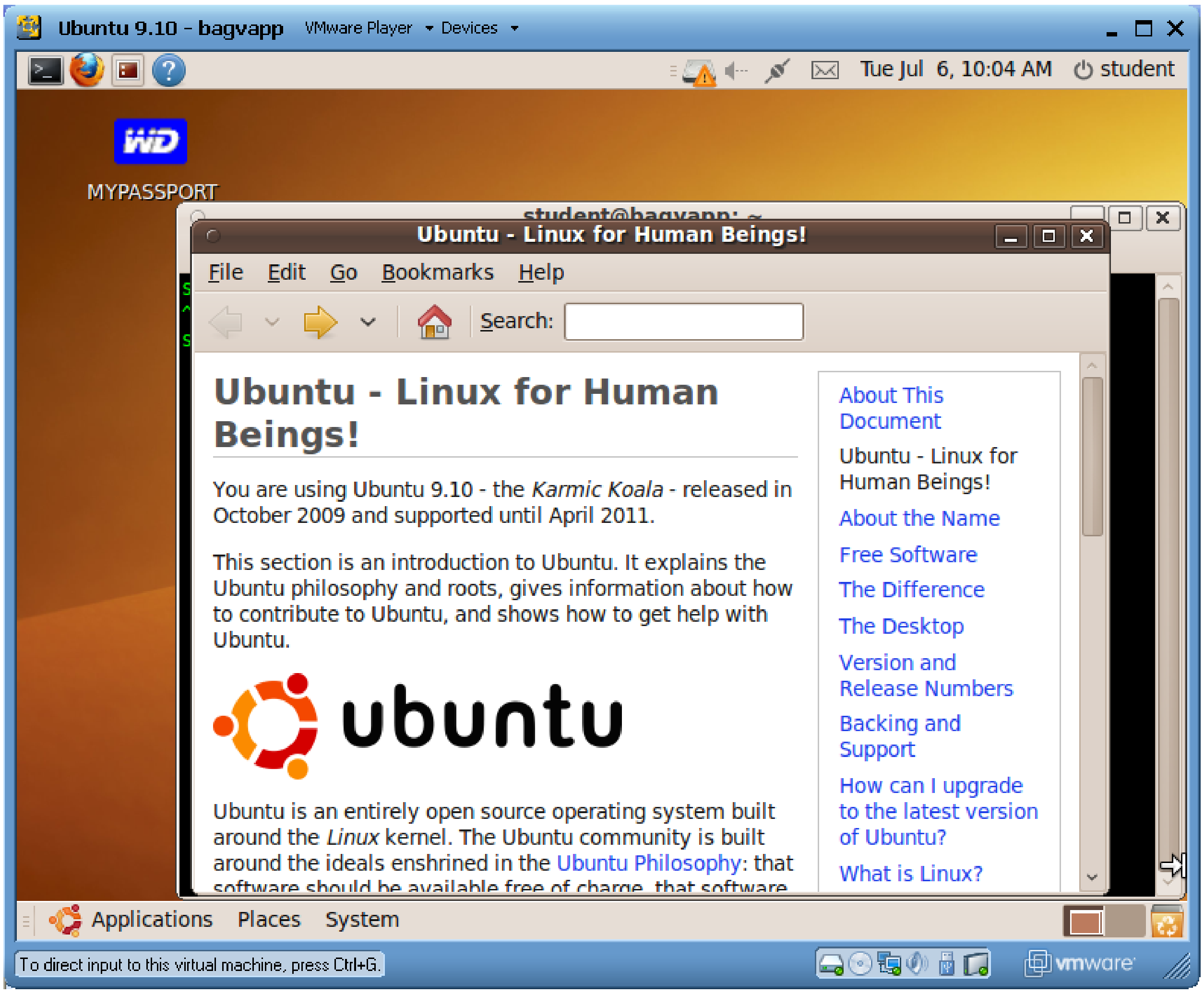
NOTE: To scroll use the middle mouse button.



```
student@bagvapp:~$ xman
```

```
XMAN is an X Window System manual browsing tool.


GETTING STARTED

By default, xman starts by creating a small window that c
"buttons" (places on which to click a pointer button).  T
buttons, Help and Quit, are self-explanatory.  The third,
creates a new manual page browser window; you may use thi
open a new manual page any time xman is running.

A new manual page starts up displaying this help informat
manual page contains three sections.  In the upper left c
menu buttons.  When the mouse is clicked on either of the
menu is popped up.  The contents of these menus is descri
Directly to the right of the menu buttons is an informati
This display usually contains the name of the directory o
being displayed.  It is also used to display warning mess
current version of xman.  The last and largest section is
information display.  This section of the application con
list of manual pages to choose from or the text of a manu

To use xman pull down the Sections menu to select a manua
When the section is displayed, click the left pointer but
of the manual page that you would like to see.  Xman will
directory listing with the manual page you selected.

That should be enough to get you started.  Once you under
basics of how to use xman, take a look at the rest of thi
```

Help topics considering Ubuntu are organized by the Ubuntu documentation project. Click on System menu and choose the option "Ubuntu help".



Ubuntu - Linux for Human Beings!

You are using Ubuntu 9.10 - the *Karmic Koala* - released in October 2009 and supported until April 2011.

This section is an introduction to Ubuntu. It explains the Ubuntu philosophy and roots, gives information about how to contribute to Ubuntu, and shows how to get help with Ubuntu.

ubuntu

Ubuntu is an entirely open source operating system built around the *Linux* kernel. The Ubuntu community is built around the ideals enshrined in the Ubuntu Philosophy: that software should be available free of charge, that software

About This Document
Ubuntu - Linux for Human Beings!
About the Name
Free Software
The Difference
The Desktop
Version and Release Numbers
Backing and Support
How can I upgrade to the latest version of Ubuntu?
What is Linux?

Command line interpreters - shells

Shells are special programs which act as command-line interpreters. Many shells are available on Linux. The default shell launched with the Ubuntu terminal window is `bash`. BASH stands for “Bourne-Again Shell” where the Bourne Shell (sh) was the first UNIX shell and today is the standard UNIX shell available on every UNIX/Linux workstation. With that, it is not widely used nowadays as there are newer and more advanced shells.

Bash is a sh-compatible command language interpreter that executes commands read from the standard input or from a file. Bash incorporated a bunch of useful features from the Korn and C shells (ksh and csh), which makes `bash` to interpret a syntax reminiscent of the programming language C.

The following table presents what can you do before hitting `Enter` button when you type a command:

Goal	Action
To move in the command line	Use the left-arrow and right-arrow keys
To insert new characters to the left of the cursor	Type new character
To delete the character to the left of the cursor	Use the Backspace or Delete key
To erase the character at the cursor	Hit Ctrl-D
To delete the characters from the cursor position to the end of the word	Hit ESC-D
To move the cursor to the beginning of the command line	Hit Ctrl-A
To move the cursor to the end of the command line	Hit Ctrl-E
To kill the command line from the cursor to the end of the line.	Hit Ctrl-K
To kill the command line to the beginning of the command line	Hit Ctrl-U

To clear the screen, puts the current command line at the top of the screen	Hit Ctrl-L
To transpose the two characters to the right and left of the cursor	Hit Ctrl-T

The structure of Linux commands

As mentioned earlier, in the paragraph “How to get some help?”, section 1 of the man-pages is the section describing general user commands.

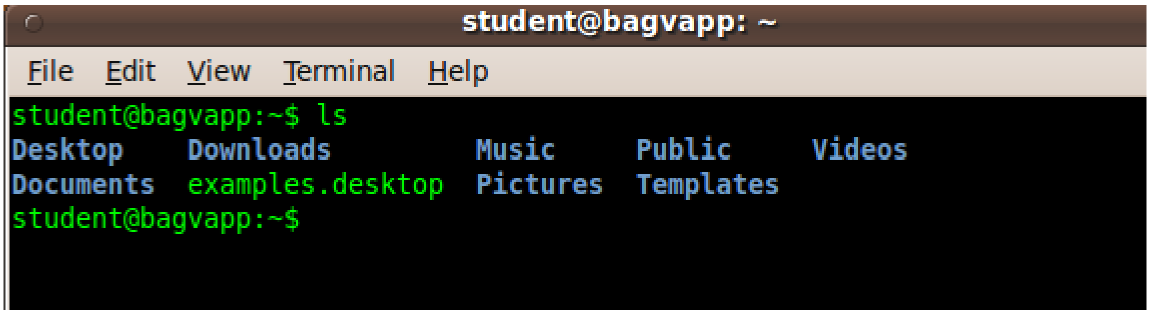
Commands are made up from a verb - the command itself, which is single word, usually consisting of lowercase letters. The verb may be followed by switches(flags) and/or arguments. Spaces separate the different components from each other.

Command switches (or flags) modify the command's action. Switches are usually preceded by a `-` sign, although there are exceptions (exceptions are out of our concern currently). Commands can have an arbitrary number of switches, and you may specify as many of them as you need (of course, you shouldn’t specify contradictory ones). For many commands, you may specify all the switches in sequence, with one `-` preceding them, and no spaces (e.g. -xyz). You may also specify each switch separately, with the `-` preceding each (e.g. -x -y -z). **Refer to the appropriate man page to find the command’s synopsis**!

Command arguments specify the entities to which the command is applied. The number of arguments a command may take can be arbitrary or fixed, and spaces separate the different arguments.

To summarize, let us look at the command `ls`. Used by itself, with no switches and arguments, it produces a list of files and folders in the current directory.

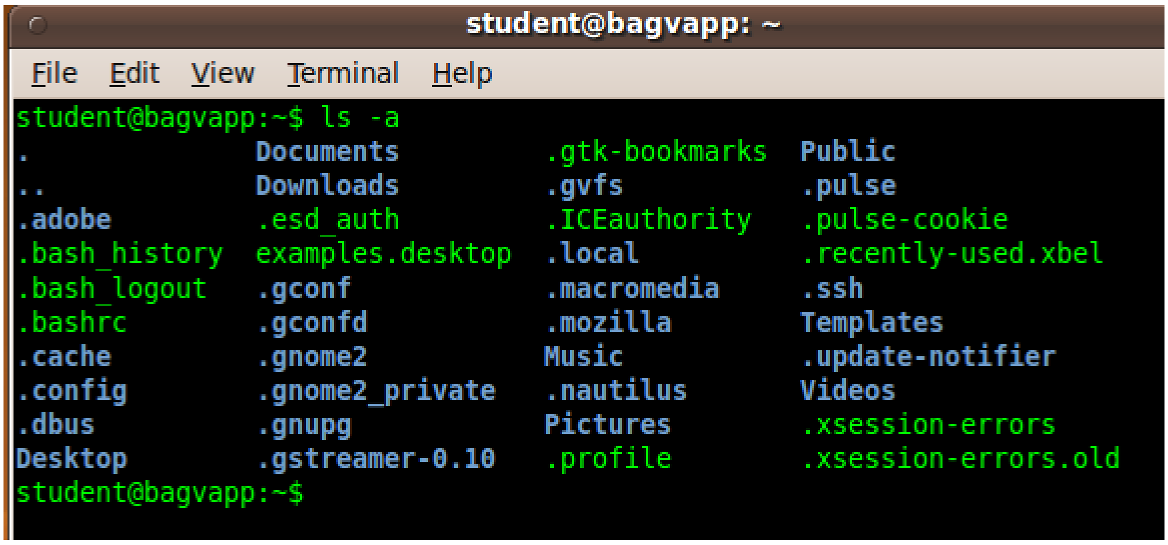
The following figure presents this list in the user’s homedir (home directory):



```
student@bagvapp:~$ ls
Desktop    Downloads         Music     Public     Videos
Documents  examples.desktop  Pictures  Templates
student@bagvapp:~$
```

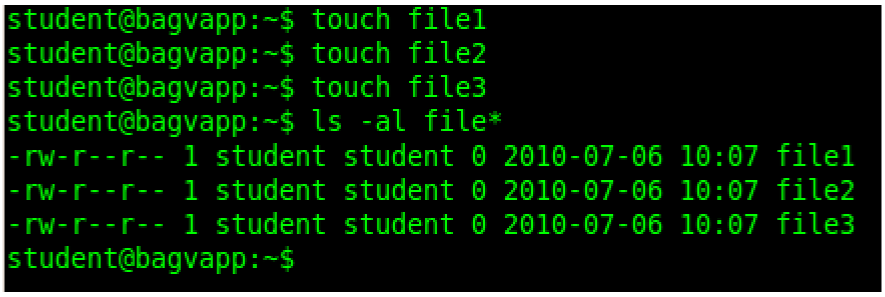
One of ‘ls’ switches is a `-a` switch, which forces printing of hidden files and hidden directories which otherwise is suppressed (hidden directory or file starts with `.`).

Following figure presents the list of all files and folders in user’s homedir:



```
student@bagvapp:~$ ls -a
.              Documents         .gtk-bookmarks  Public
..             Downloads         .gvfs           .pulse
.adobe         .esd_auth         .ICEauthority   .pulse-cookie
.bash_history  examples.desktop  .local          .recently-used.xbel
.bash_logout   .gconf            .macromedia     .ssh
.bashrc        .gconfd           .mozilla        Templates
.cache         .gnome2           Music           .update-notifier
.config        .gnome2_private   .nautilus       Videos
.dbus          .gnupg            Pictures        .xsession-errors
Desktop        .gstreamer-0.10   .profile        .xsession-errors.old
student@bagvapp:~$
```

The following figure shows an example of “ls” command using arguments and flag concatenation. “file*” is a ‘wildcard’ argument. To learn more please refer to the paragraph “Wildcards” and “-al” is equal to “-a -l”. Files “file1”, …,”file3” were created by command “touch <fileN>”):



```
student@bagvapp:~$ touch file1
student@bagvapp:~$ touch file2
student@bagvapp:~$ touch file3
student@bagvapp:~$ ls -al file*
-rw-r--r-- 1 student student 0 2010-07-06 10:07 file1
-rw-r--r-- 1 student student 0 2010-07-06 10:07 file2
-rw-r--r-- 1 student student 0 2010-07-06 10:07 file3
student@bagvapp:~$
```

The file system

Under Linux, all files are organized in one file system in a tree-like structure. Files can be grouped together in directories (or folders), which themselves are grouped together in other directories, And finally everything is grouped together under the file system root, which is denoted by the symbol /.

The directories `bin', `etc', `tmp`, `var`, `dev`, `home` and `usr' are standard Linux directories containing system files, user accounts etc. The location of user accounts, and therefore user files and directories is chosen by the system administrator. On the Ubuntu user accounts are located under the directory `/home/' which has subdirectories named by the users' names. So, user named `student` has a homedir named "student" and its full path is /home/student.

Following is a table of commands related to file system traversal:

Goal	Command	Example
To move from directory to directory (".." stands for the parent directory)	cd	cd ..
To print the full name of working directory	pwd	pwd
To list the contents of working directory	ls	ls ~student

Note: `~<username>` is shorthand of `/home/<username>` (e.g. `~student` = `/home/student`).

Wildcards

Linux shells let you use wildcards as shortcuts in specifying file names. Wildcards are special characters which are used to group many files together:

- The * matches zero or more characters. For instance, `ls *` will display all the files in the current directory, and `ls a*b` will display all the files whose names start with an `a`, and end with a `b` (including the file `ab`, if it exists).
- The ? matches a single character. For instance, `ls a?b` will display all the files whose names are 3-characters long, starting with an `a` and ending with a `b`, and `ls ?????` will display all files whose names are exactly 5-characters long.
- The [ ] match any single character from the set specified inside the brackets. Thus, `ls [abc]d[abc] will display all files whose names are exactly 3 characters long, have an a,b or c as the first and last character in their name, and a d as the middle character. If you want to match a consecutive range of letters or digits, you may use a `-` to specify the range. For example, `ls [a-z]*[0-9]` will display all files whose names are at least 2-characters long, which start with a lowercase letter, and end with a digit.

Note, that on the Linux systems the period (`.`) in the middle of the file (directory) name has no special significance (to stress this fact, note that even executable files on Linux/UNIX systems have no `.exe` extension). Therefore, wildcards will also match the period.

Several file-related commands

The reader is invited to look for more information in the man pages:

Goal	Command
To print the contents of the file onto standard output	cat
To print the contents of the file to standard output allowing only forward movement (by hitting the `Enter` or `space`)	more
To print the contents of the file onto standard output allowing both forward movement (`Enter` or `space`) and backward movement (Ctrl-`up arrow`)	less
To copy files	cp
To move files	mv
To remove all files whose names were specified as arguments	rm
To create a new file	touch
To change the modification date of files	touch
To compare between two files	diff
To find the location of a file in the file system	find
To search for a pattern in a file	grep
To display the first few lines of the specified file	head
To display the last few lines of the specified file	tail

Caution: Be careful when deleting files. There is no way to restore them afterwards.

Several directory-related commands

The reader is invited to look for more information in the man pages:

Goal	Command
To copy one directory onto another	cp
To move one directory onto another	mv
To print the contents of the directory onto standard output	ls
To remove empty directory	rmdir
To remove non-empty directory	rm –r
To create a new directory	mkdir
To fine directory in the file system	find

Caution: Be careful when deleting directories. There is no way to restore them afterwards.

Redirecting Input/Output

By default, shell commands take their input from the standard input (which usually is the keyboard), and direct their output to the standard output (usually the terminal's screen). You may modify this behavior by using the <, > and >> symbols.

The < defines a file to be used instead of the standard input. The > defines a file to be used instead of the standard output. Every time you use the > symbol, the file is recreated, i.e. its previous contents is lost. If you want to append new data to an existing file, use >> instead.
For example, typing `ls –al > file_list` causes to file file_list to be crated/recreated and the output of `ls –al` to be positioned therein.

Bash provides one more way of I/O redirection called pipes. Pipes are used to concatenate two or more commands, so that the output of one command will be the input of the next command. Pipe-line symbol `|` is used to indicate shell that pipes are used.

For example, `ls | sort` will sort the output of ls command alphabetically, saving keystrokes and eliminating the necessity of creating intermediate files when no pipes were used.

Another example demonstrates the way to count the number of words in some given file:
`cat some_file | wc –w`.

Protecting files and directories, altering protections

Protection scheme in native Linux file system applies to three categories of populations:

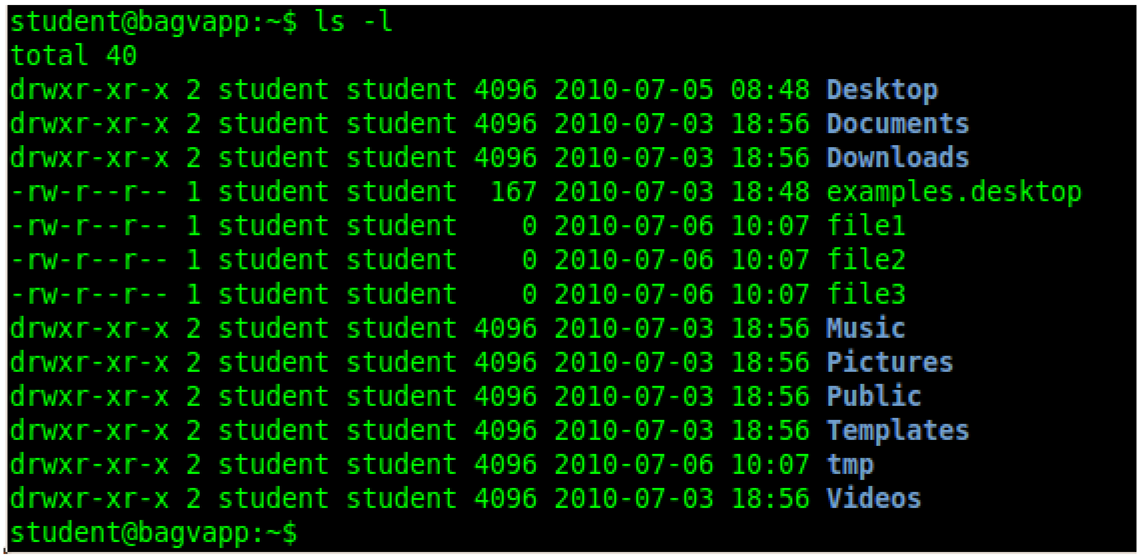
Notation of population	Description of population
u	the user's (or owner's) protections – reflects the state of protection 'owner vs. everybody else'.
g	the owner's group protections – reflects the state of protection 'owner vs. users that belong to the groups that the owner belongs to also'. Group of users may be a group of all 3rd-year students, defined by system administrator.
o	all other users protections – reflects the state of protection 'owner vs. everybody else'

There are three basic categories of protections:

Notation of category	Description of category
R	for read protection. This protection determines whether a user may read a file or directory, and get a list of the files in a directory.
W	for write protection. This protection determines whether a user may write to a file , and create or delete files in a directory.
X	for execute protection. This protection determines whether a user may execute (run) a

	file (assuming it an binary executable or script file), and whether a user may perform `cd` command to a directory.

For example, see the following figure listing the contents of folder ~student :



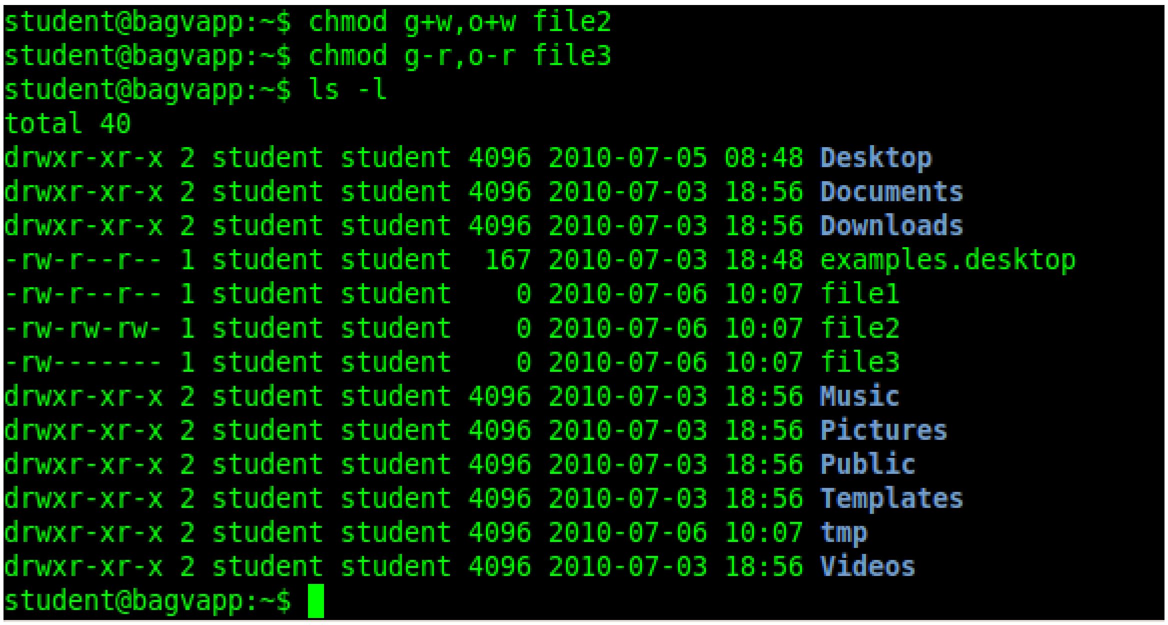
```
student@bagvapp:~$ ls -l
total 40
drwxr-xr-x 2 student student 4096 2010-07-05 08:48 Desktop
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Documents
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Downloads
-rw-r--r-- 1 student student  167 2010-07-03 18:48 examples.desktop
-rw-r--r-- 1 student student    0 2010-07-06 10:07 file1
-rw-r--r-- 1 student student    0 2010-07-06 10:07 file2
-rw-r--r-- 1 student student    0 2010-07-06 10:07 file3
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Music
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Pictures
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Public
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Templates
drwxr-xr-x 2 student student 4096 2010-07-06 10:07 tmp
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Videos
student@bagvapp:~$
```

The first character on each line is usually the `-` for a file and a `d` for a directory. The next 9 characters specify the protections, and are grouped in 3 groups of 3 characters each, for the user (=owner), group and other protections. Thus in the example above, `file1` is a file, whose owner (`student`) has read and write privileges on it, and the group and others have only read privileges on it. That means that the user ` student ` may read the file and modify it, and everybody else may only read it. The directory `tmp' has the `rwx' bits set for the owner. Therefore, ` student ` may see a list of files in “tmp” (because of `r`), may create and delete files there (because of `w`) and may cd to `tmp` (because of `x`). All other users (including the group) may see a list of files, may make a “cd” to the `tmp` directory, but may not modify it.

NOTE: The protection mechanism has more than 3 basic schemes. You are invited to refer to the `chmod`, `chgrp`, `chusr` commands’ man-pages.

chmod command

To change a file or directory protections one can use the command `chmod`. For example, to give a write permission to everybody for the file2 and to revoke the read permission for everybody except the owner of file3, enter the command `chmod g+w,o+w file2` and immediately afterwards the enter the command `chmod g-r,o-r file3`. The result is displayed bellow:



```
student@bagvapp:~$ chmod g+w,o+w file2
student@bagvapp:~$ chmod g-r,o-r file3
student@bagvapp:~$ ls -l
total 40
drwxr-xr-x 2 student student 4096 2010-07-05 08:48 Desktop
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Documents
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Downloads
-rw-r--r-- 1 student student  167 2010-07-03 18:48 examples.desktop
-rw-r--r-- 1 student student    0 2010-07-06 10:07 file1
-rw-rw-rw- 1 student student    0 2010-07-06 10:07 file2
-rw------- 1 student student    0 2010-07-06 10:07 file3
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Music
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Pictures
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Public
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Templates
drwxr-xr-x 2 student student 4096 2010-07-06 10:07 tmp
drwxr-xr-x 2 student student 4096 2010-07-03 18:56 Videos
student@bagvapp:~$
```

For more information about the chmod command the reader is invited to look in the man pages.

PostScript and PDF file viewer

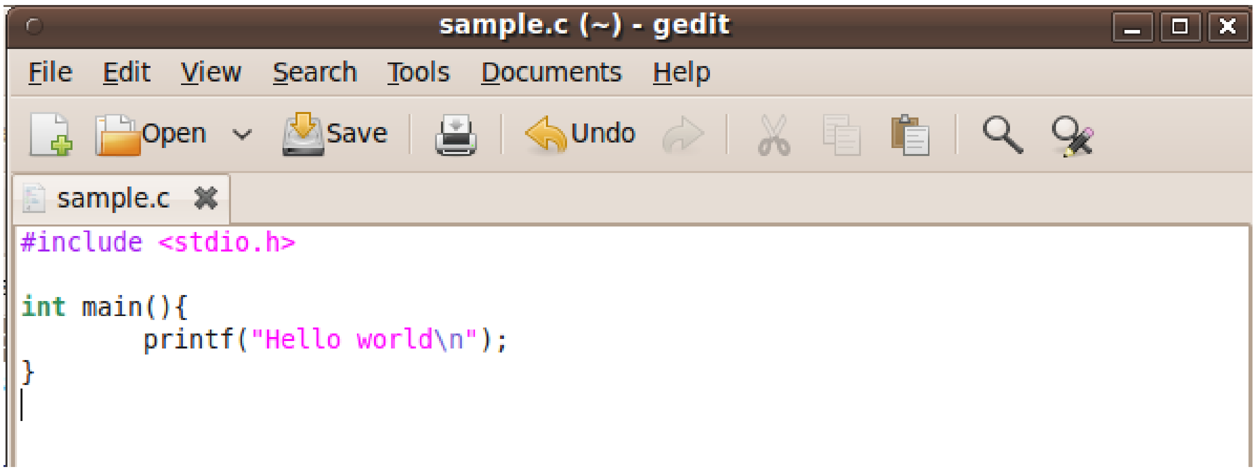
Ubuntu comes with several viewers for PostScript (*.ps) and PDF (*.pdf) files. But ,probably, the most friendly one is a *evince* previewer. Launch "*evince* &" from the terminal.

Editors

Find experimentally the editor that is most suitable for you (from the Applications menu). This manual briefly relates some of them.

gedit editor

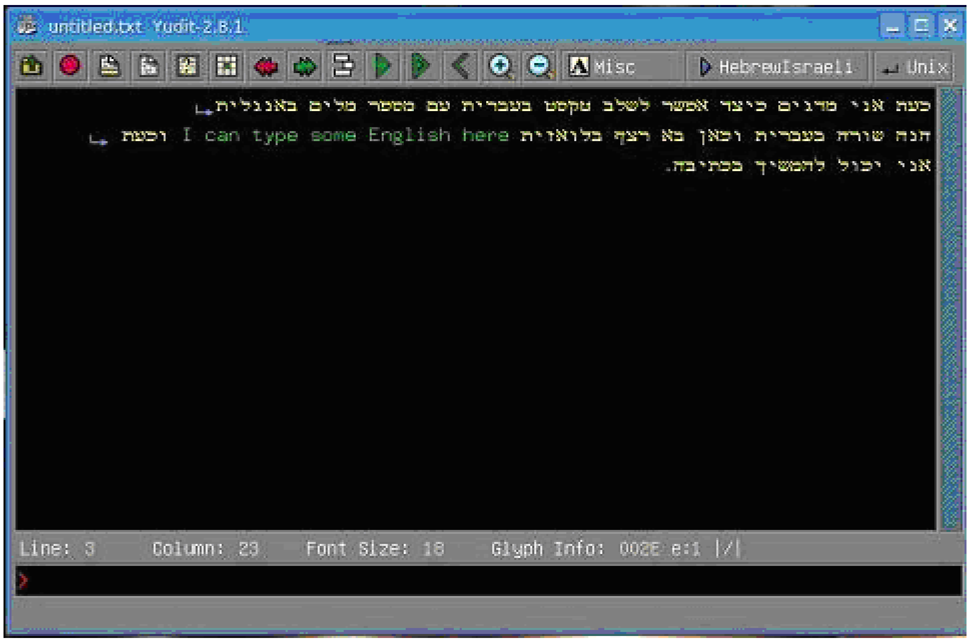
gedit editor is a simple and self explanatory text editor that supports automatic highlighting any *.c or *.h files. *gedit* is launched by typing "*gedit* &" from the terminal:



```
#include <stdio.h>

int main(){
        printf("Hello world\n");
}
```

yudit editor

yudit is a simple Unicode text editor for X Window System. To invoke the *yudit* editor type “yudit&” from the terminal. Using *yudit* is pretty straightforward. To change keyboard map from English to Hebrew use familiar ALT-Shift combination.



```
כעת אני מדגים כיצד אפשר לשלב טקסט בעברית עם מספר מלים באנגלית
הנה שורת בעברית וכאן בא רצף בלואזית I can type some English here וכעת
אני יכול להמשיך בכתיבה.
```

```
Line: 3     Column: 23     Font Size: 18     Glyph Info: 002E e:1 |/|
```

Emacs editor

Emacs is a complete working environment, rather then just a text editor. Among many other features Emacs has programming modes that support an:

- automatic color-syntax highlighting for programming languages
- automatic indentation and code formatting
- interfacing with ones favorite debugger
- addition of language-specific menus to the Emacs menu bar

Properly configured, Emacs is quite like a commercially marketed Integrated Development Environments for programming languages like C++ and Java. But configuring Emacs is a subject to be covered in a different manual. **Beginners are better to use Emacs as a code editor with automatic support of programming languages highlighting, indentation and formatting.**

Emacs is invoked by the command emacs (type “emacs &” from the terminal). Graphical user interface provides elaborate menu. But beginners may find emacs to be a little bit nonhabitual in several ways. So, reading “Emacs Tutorial” that is availabe from the program’s Help menu might be useful.

Vi editor

Vi is a very powerful editor found on every Unix/Linux machine. Ubuntu comes with enhanced version that is backward compatible with vi and called vim (vi improved). But beginners may discover it somehow tedious to use non-graphical text processor, however powerful is it. Vi supplies features that no other editor familiar so far can supply. Bellow is a summary intended to provide only a basic outline (everything is referred to VISUAL mode, unless a preceding colon ‘:’ is specified):

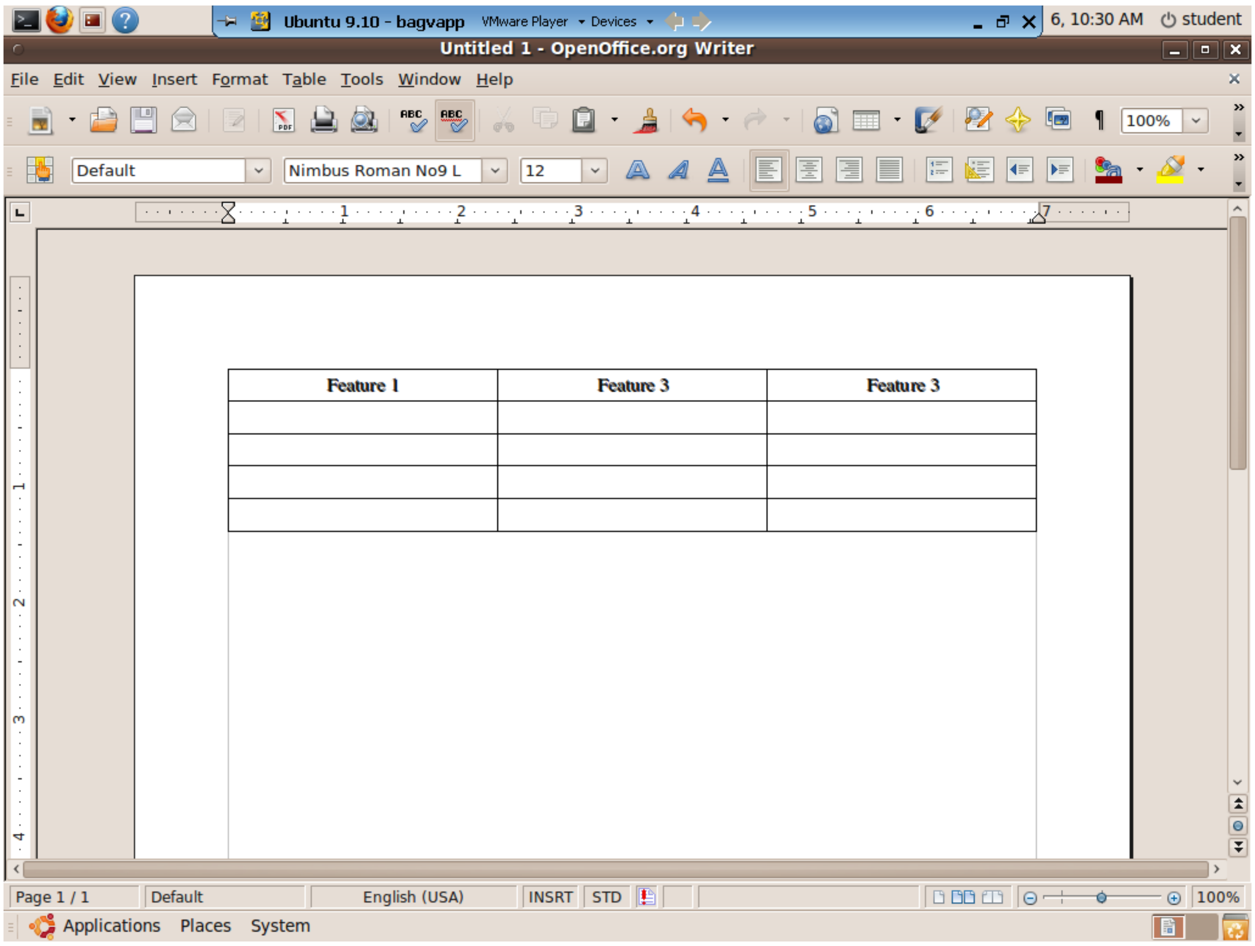
Invocation	
vi file	Edit at first line of 'file'
Exiting	
:wq	Write buffer, then quit.
:w	Write buffer to original file
:w file	Write buffer to 'file'
:w>> file	Append to 'file'
:q	Quit out of editor
:q!	Quit discarding changes
Text input (ESCape key terminates Input mode)	
i	Insert before cursor
Insert Mode Commands	
Control-@	Repeat Last Insertion
Control-W	Erase last word entered
DELete or Backspace	Same as Control-H

Control-U	Erase entire line
Cursor Motion	
down-arrow or j	Move cursor down one line
up-arrow or k	... up one line, staying in same column
left-arrow or h	... to the left
right-arrow or l	... to the right
Substitution	
:s/X/Y/opt	Substitute 'Y' for first occurrence of 'X' Options: g - Change every occurrence in line c - Confirm each change p - Print each change (Command mode only)
:g/X/s//Y/opt	Globally find the first 'X' on each line and substitute 'Y' for the string 'X'
Deleting Text	
x	Delete character under cursor
nx	... 'n' characters
dd	... entire line
D	... to end of line
Undo	
u	Undo last change
U	Restore current line

Open Office

Open Office is the open-source office software suite for word processing, spreadsheets, presentations, graphics, databases and more. It is available in many languages and works on all common computers. It stores all your data in various formats and can also read and write files from other common office software packages (e.g. MS Word).

Note: To change keyboard map from English to Hebrew use familiar ALT-Shift combination.



Feature 1	Feature 3	Feature 3

The gcc compiler

gcc is a program used to compile C programs. Process of compilation consists of 3-steps: preprocessing, compilation and linkage. In the first stage, the user's program is preprocessed. During that stage all preprocessor directives are fulfilled and preprocessed code files are prepared for the second stage – a compilation. Compilation is a translation from the programming language to the machine language. During this stage an intermediate files, called an object files, are created. The last stage is a linkage, standing for the linkage of created object files in an executable program.

In many cases this 3-step process is transparent for the user. For example a command `gcc sample.c` creates executable named a.out and combines all above steps. But while writing big projects using many files, we can find it useful to split the steps of compilation. For more details see the paragraph "Make utility".

Following table lists several gcc–related switches. For detailed information refer to man-pages:

Switch	Example	Description
-c	gcc –c sample.c	Creates object file sample.o (proceeding the preprocessing and compilation steps)
-o	gcc sample.c –o sample	Creates executable file with the name sample, instead of the default `a.out'
-Wall	gcc –Wall sample.c	Displays compilation warnings that were suppressed otherwise
-g	gcc –g sample.c	Links debugging libraries into the executable which is necessary for further debugging.

Combining all above results in 'gcc –Wall –g sample.c –o sample' directing the 'gcc' to create executable file named 'sample' that incorporates debugging libraries while compiler doesn't suppress compilation warnings.

There are lots of options for 'gcc', which are all in the man page. Here are a few of the advanced ones, with examples of how to use them:

Switch	Example	Description
-ansi	gcc –ansi sample.c	Check that your code complies to the relevant international standard, often referred to as the ANSI standard, though strictly speaking it is an ISO standard. One of the main aims of the standard is to allow people to write code that will work with any compiler on any system. This is known as portable code. ‘-ansi’ turn off most, but not all, of the non-ANSI C features provided by ‘gcc’`. Despite the name, it does not guarantee strictly that your code will comply to the standard
-pedantic	gcc -Wall -ansi -pedantic sample.c –o sample	‘-pedantic’ turns off all ‘gcc’ 's non-ANSI C features. This example produces an executable ‘sample’ after checking ‘sample.c’ for standard compliance. Generally, you should try to make your code as portable as possible, as otherwise you may have to completely re-write the program later to get it to work somewhere else - and who knows what you may be using in a few years time?
-O	gcc -O sample.c –o sample	‘-O’ creates an optimized version of the executable. The compiler performs various clever tricks to try and produce an executable that runs faster than normal. You can add a number after the ‘-O’ to specify a higher level of optimization. Optimization is usually only turned on when compiling a release version.

Libraries

Libraries provide a way of organizing modules into larger entities. We often need a way to make a collection of modules (e.g. object files), that provide common functionality, things that many programs will want to do – working with files, for example. The pieces that do these things may be provided for you in the form of ‘static’ or ‘dynamic’ libraries. Library can be thought as a super-object file, providing collection of capabilities with certain functionality (functions, classes, etc.). Libraries are usually created from a set of objects. Library interface (C/C++ header files, Java API docs, ...) is often provided to the developers together with the binary library.

The difference between static and dynamic libraries is in the type of linking. Static libraries are physically copied into the executable file, while dynamic liberties are loaded on demand. Dynamic libraries are often called ‘shared libraries’ because library files are usually shared by different programs thus saving disk space by reducing the size of executable files.

Static libraries have names like libname.a. They are created running from the command line:

```
ar rcu libname.a object1.o object2.o ...
ranlib libname.a
```

'ar' utility simply 'archives' the .o files into the repository named 'libname.a'. The suffix .a comes from ‘archive’. 'ranlib' utility goes over the archive and extracts the symbols tables from individual .o files writhing the archive and builds a large symbol table for all the library. Symbol table contains names (function names, global variables' names, ...) and the position of the machine code implementing them. Linker uses this information when producing executables to find the code of each function and to check whether the global variables are defined.

Suppose, files ‘my_read.c’, ‘my_write.c’ contain library functions for I/O. Following lines create static library for I/O:

```
gcc –Wall –c my_read.c

gcc –Wall –c my_write.c

ar rcu libmy_io.a my_read.o my_write.o

ranlib libmy_io.a
```

And finally, one should link all object files and libraries altogether. Again, suppose that 'module1.c',…, 'moduleN.c' are compiled and some of them are using I/O library functions. Then the following line compiles 'driver.c' that contains 'main()' and links program modules and libmy_io.a into the executable named 'driver':

```
gcc –Wall –L. driver.c module.o module2.o … moduleN.o –lmy_io –o driver
```

'–l**my_io'** stands for the library 'lib**my_io**.a' and the search for the file 'libmy_io.a' is made in the current directory which is indicated by the '-L.' flag ('.' stands for the current directory).

Shared libraries have names like `libname.so.x.y.z` where `x.y.z` is a form of version number. They are created running from the command line:

```
gcc –fPIC –c file1.c
…
gcc –fPIC –c fileN.c
gcc –shared libname.so file1.o file2.o … fileN.o
```

'gcc –fPic' or '-fpic' directs to compile for "Position Independent Code". When object files are generated, we have no idea where in memory they will be inserted in a program that will use them. Many different programs may use the same library, and each load it into a different memory in address. Thus, we need that all jump calls ("goto", in assembly speak) and subroutine calls will use relative addresses, and not absolute addresses. Thus, we need to use a compiler flag that will cause this type of code to be generated. The first three commands compile the source files with the 'fPIC' option, so they will be suitable for use in a shared library (they may still be used in a program directly, even thought they were compiled with 'fPIC').

'gcc –shared' used to generate shared library, and directs compiler (or linker) that it should relate a shared library, not a final program file.

The compilation part is easy. It is done almost the same as when linking with static libraries:

```
gcc –Wall  driver.c -L. -lname -o driver
```

The linker will look for the file 'libutil.so' (-lutil) in the current directory (-L.), and link it to the program, but will not place its object files inside the resulting executable file ‘driver’.

If both a static (lib**name**.a) and dynamic (lib**name**.so) libraries are present in the current directory, the default behaviour is to link the shared library). To force the linker to link statically you need to specify the ‘-static’ option.

The make utility

As mentioned in previous paragraph, sometimes it is useful to deliberately split the compilation process into separate stages. For example, suppose we wrote a text editor of ten thousand lines of code that spread over the twenty source code files. It worth only to compile files first to produce an object files and then, as a final stage, to link them to produce an executable file. Doing it this way we can significantly reduce the time of consequent compilations when some of a source files will be changed, avoiding a compilation of an unchanged source files.

In many cases, especially when working with big projects, another dimension is added to the compilation process: we need to produce a number of targets (for example a number of executables or a shared libraries) and it seems to be very tedious to check every source file whether it was changed or whether not just to save a time of recompilation. To optimize this process utility called a make utility was created. Make utility automatically determines modified source files concluding that corresponding object files (libraries, executables) need to be updated and issues commands to update them.

To use the ‘make’ utility one have to write a structured file (often called a makefile) that describes a relationships among your source files in each of your target programs and states commands generating those targets.

Once a suitable makefile exists, each time you change some source files, running `make` command from the command line suffices to perform all necessary recompilations. The make utility uses the makefile data and files' last-modification times to decide which of the files need to be updated. For each of those files, 'make' utility issues commands stated in the makefile.

As was mentioned, makefiles are structured files. As a matter of fact, elements of that structure form a versatile scripting language, but we'll see only some of those elements:

Scripting language element	Description
file.o: file.c utils.c file.h gcc –c –g file.c utils.c	This is an example of explicit rule which states that the **target** file.o depends on **dependencies** file.c, utils.c and file.h and the **command** to produce the target is `gcc –c –g file.c utils.c`. **NOTE, that the command should start with TAB in the beginning of the command line. If TAB is missing or space keys are stroked instead of TAB, 'make' utility stops with error message:** **"Makefile:line#**Missing separator. Stop"**

libutils.a: utils1.o utils2.o
    ar rcu libutils.a utils1.o utils2.o
    ranlib libutils.a
	This is an example of explicit rule with target that needs two commands to be issued. In general, there may be more then one command line.

SRS_FILES = file1.c file2.c file3.c
 	
ex1: $(SRC_FILES)
    gcc –o ex1 $(SRC_FILES)
	This is an example of variable definition and its usage. We defined a SRC_FILES variable and the consequent use of that variable saves us several keystrokes and is seems to be also less error-prone.
	$(SRC_FILES) is a directive to use the value of variable SRC_FILES.

# ***************************** # this makefile is last modified at 2/3/90 #******************************	\`#\` starts a comment which spans until end of the line

ex1: file1.c file2.c file3.c
        gcc –Wall –o ex1 file1.c \
                file2.c file3.c
	\`\\\` indicates that the next line is a continuation of the current command

all: my_server my_client my_server: server.c gcc server.c –o my_server my_client: client.c gcc client.c –o my_client	This is an example of phoney node usage. Phoney nodes are special targets that have no commands followed by. The only target that make utility eventually produces is the first target that is bumped into when parsing the makefile. While executing, a make utility builds a tree of target dependences according to supplied makefile with the first target located in the root of a dependency tree. Those targets that were not included in the dependency tree are silently disregarded. The \`all\` phoney node prompts the make utility to generate several dependency trees. In our example, one for the target my_server and another one for the target my_client.
clean: rm *.o ex1 *~	Running 'make clean' from the command line invokes 'rm -f *.o ex1 *~' forcing ('-f' flag) all files with suffix '.o', the file named executable 'ex1' and all files with suffix '~' to be deleted. Files with suffix '~' are old copies saved by the Emacs editor.

Let's see now a demonstration of to how put all those elements together. Suppose, files celsius.h, celsius.c, kelvin.h, kelvin.c, fahrenheit.h, fahrenheit.c provide functions to convert to Celsius, Kelvin and Fahrenheit representations accordingly from the remaining two. Files convert_celsius.c, convert_kelvin.c and convert_fahrenheit.c contain each a ‘main()’ function. An argument that main() receives is a temperature in Celsius, Kelvin and Fahrenheit representation accordingly. The output is a temperature converted to the other two representations. You can see the source code in “Makefile Examples” directory (the directory was archived with this manual). Let’s see three different ways to write a makefile generating executables ‘convcelsius’, ‘convkelvin’ and ‘convfahrenheit’.

Example 1:

```
#simple Makefile with all the dependencies explicitly listed

all: convkelvin convfahrenheit convcelsius

convfahrenheit: celsius.o kelvin.o convert_fahrenheit.o
	gcc -o convfahrenheit celsius.o kelvin.o \
		convert_fahrenheit.o

convcelsius: kelvin.o fahrenheit.o convert_celsius.o
	gcc -o convcelsius kelvin.o fahrenheit.o \
		convert_celsius.o

convkelvin: celsius.o fahrenheit.o convert_kelvin.o
	gcc -o convkelvin celsius.o fahrenheit.o \
		convert_kelvin.o

celsius.o: celsius.c celsius.h
	gcc -c celsius.c

kelvin.o:  kelvin.c  kelvin.h
	gcc -c kelvin.c

fahrenheit.o: fahrenheit.c fahrenheit.h
	gcc -c fahrenheit.c

convert_kelvin.o: convert_kelvin.c fahrenheit.h celsius.h
	gcc -c convert_kelvin.c

convert_fahrenheit.o: convert_fahrenheit.c kelvin.h celsius.h
	gcc -c convert_fahrenheit.c

convert_celsius.o: convert_celsius.c fahrenheit.h kelvin.h
	gcc -c convert_celsius.c

# cleanup
clean:
	rm *.o *~
```

Example 2

```
all: convkelvin convfahrenheit convcelsius


# $@ and $^ are 'automatic variables'. They are updated before
# the command is issued. The value of $@ is set to the target
# and the value of $^ is set to the list of all the dependencies
# of the rule

convfahrenheit: celsius.o kelvin.o convert_fahrenheit.o
	gcc -o $@ $^

convcelsius: kelvin.o fahrenheit.o convert_celsius.o
	gcc -o $@ $^

convkelvin: celsius.o fahrenheit.o convert_kelvin.o
	gcc -o $@ $^


# '%.o: %.c' is a 'pattern rule' telling the 'make' utility to
# search for files with suffix '.c' and create targets
# with the same name replacing '.c' by '.o'.
# $< is an automatic variable that holds the name of the current
# dependency.
%.o: %.c
	gcc -c $<

# cleanup
clean:
	rm *.o *~
```

Example 3

```
# 'make' utility allows variable definitions. For example,
# the value of the variable CC is set to the name of compiler
# and the value of HDRS is set to the list of header files
CC	= gcc
CFLAGS		= -Wall
LFLAGS		= -L.
LIBS	= -lm
HDRS	= celsius.h fahrenheit.h kelvin.h
OBJS_convfahrenheit	= celsius.o kelvin.o convert_fahrenheit.o
OBJS_convcelsius	= kelvin.o fahrenheit.o convert_celsius.o
OBJS_convkelvin		= celsius.o fahrenheit.o convert_kelvin.o


# $(VAR) is a directive to use the value of variable VAR defined before
OBJS	= $(OBJS_convfahrenheit) $(OBJS_convcelsius) $(OBJS_convkelvin)
```

```
# 'OBJS:.o=.c' is a construction that directs "take the members of the list
# OBJS and create a new list substituting .o  by .c"
SRCS  = $(OBJS:.o=.c)


all: convkelvin convfahrenheit convcelsius

convfahrenheit: $(OBJS_convfahrenheit)
	$(CC) -o $@ $^ $(LFLAGS) $(LIBS)

convcelsius: $(OBJS_convcelsius)
	$(CC) -o $@ $^ $(LFLAGS) $(LIBS)

convkelvin: $(OBJS_convkelvin)
	$(CC) -o $@ $^ $(LFLAGS) $(LIBS)

%.o: %.c
	$(CC) $(CFLAGS) -c $<

# phony nodes
clean:
	rm *.o *~ *.bak
```

Debugging your program

Ubuntu comes with a standard GNU gdb debugger. Please refer to info pages to learn about the gdb debugger as a standalone program or to get acquainted of how to run the gdb from emacs. To run the gdb debugger type ‘gdb’ from the command line. It worth also to mention, that we supplied Ubuntu with Eclipse Integrated Development Environment which is parallel to the Microsoft Visual Studio. We will not cover Eclipse in this brochure.

This chapter will teach you an additional way: invoking gdb using the graphical front-end named DDD.

Guidance will be based on the sample program ‘sample.c’. Create in your homedir (~student) a folder named ‘sample’. Copy to that folder files ‘Makefile’ and ‘sample.c’ (files are located in the ‘~student/courses/os’ directory).

Here's the source sample.c of the sample program.

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

static void shell_sort(int a[], int size)
{
   int i, j;
   int h = 1;
   do {
      h = h * 3 + 1;
   } while (h <= size);
   do {
      h /= 3;
      for (i = h; i < size; i++)
      {
         int v = a[i];
         for (j = i; j >= h && a[j - h] > v; j -= h)
            a[j] = a[j - h];
         if (i != j)
            a[j] = v;
      }
   } while (h != 1);
}

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

```
{
   int *a;
   int i;

   a = (int *)malloc((argc - 1) * sizeof(int));
   for (i = 0; i < argc - 1; i++)
      a[i] = atoi(argv[i + 1]);

   shell_sort(a, argc);

   for (i = 0; i < argc - 1; i++)
      printf("%d ", a[i]);
   printf("\n");

   free(a);
   return 0;
}
```

The sample program ‘sample.c’ exhibits the following bug. Normally, sample should sort and print its arguments numerically, as in the following example:

```
$ ./sample 8 7 5 4 1 3
1 3 4 5 7 8
$ _
```

However, with certain arguments, this goes wrong:

```
$ ./sample 8000 7000 5000 1000 4000
1000 1913 4000 5000 7000
$ _
```

Although the output is sorted and contains the right number of arguments, some arguments are missing and replaced by bogus numbers; here, 8000 is missing and replaced by 1913.

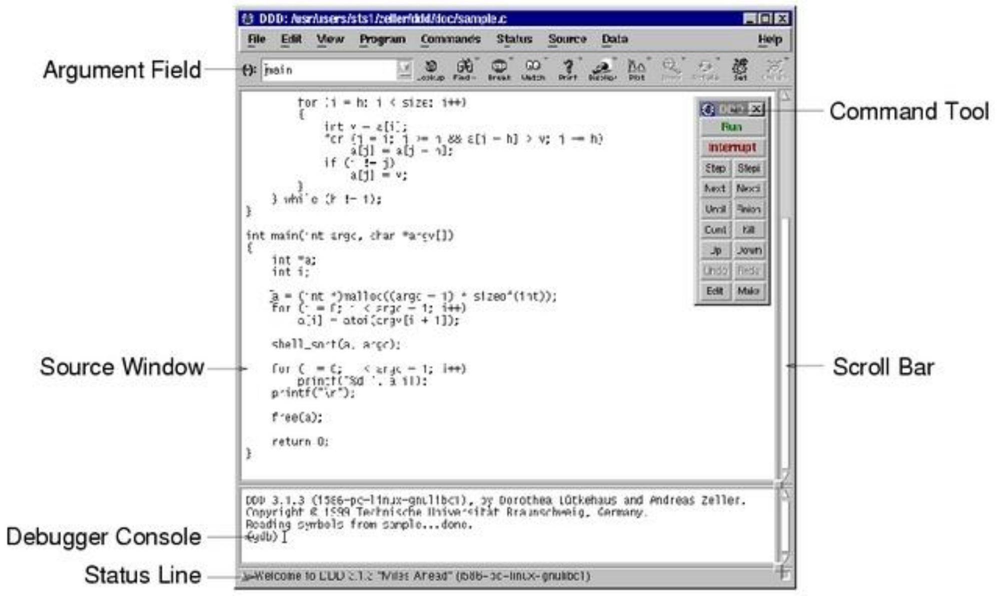
Let us use DDD to see what is going on. First, you must compile sample.c for debugging. Run the make utility from the command line:

```
$ make
```

Now, you can invoke DDD on the sample executable:

```
$ ddd sample
```

After a few seconds, DDD comes up. The *Source Window* contains the source of your debugged program; use the *Scroll Bar* to scroll through the file.



```
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

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

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
```

```
DDD 3.1.3 (i586-pc-linux-gnulibc1), by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1999 Technische Universität Braunschweig, Germany.
Reading symbols from sample...done.
(gdb) 
```

```
Welcome to DDD 3.1.2 "Man Ahead" (i586-pc-linux-gnulibc1)
```

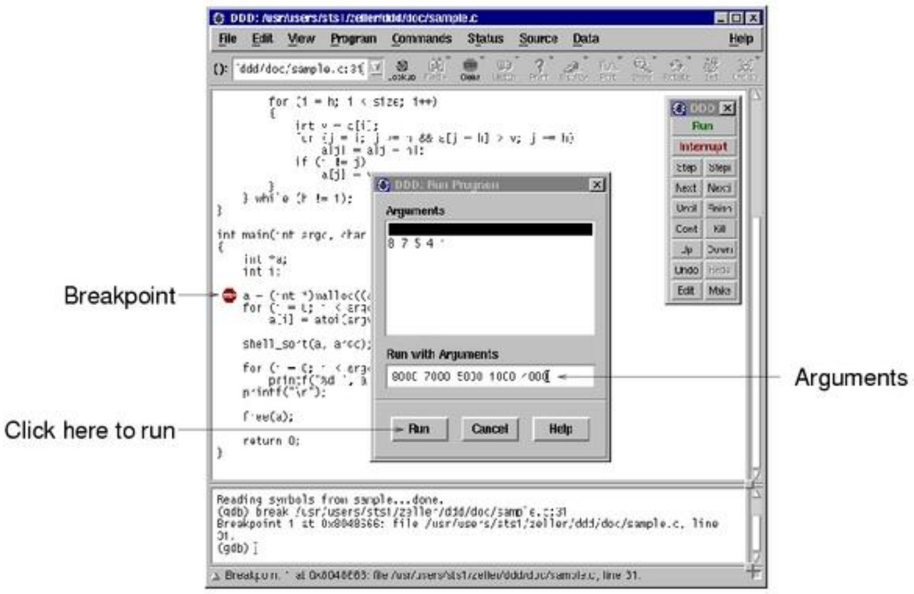
Initial DDD Window

The *Debugger Console* (at the bottom) contains DDD version information as well as a GDB prompt.

```
GNU DDD Version 3.3.9, by Dorothea L□tkehaus and Andreas Zeller.
Copyright © 1995-1999 Technische Universitהt Braunschweig, Germany.
Copyright © 1999-2001 Universitהt Passau, Germany.
Copyright © 2001-2004 Universitהt des Saarlandes, Germany.
Reading symbols from sample...done.
(gdb) _
```

The first thing to do now is to place a *Breakpoint* , making sample stop at a location you are interested in. Click on the blank space left to the initialization of a. The *Argument field* (): now contains the location (sample.c:31). Now, click on Break to create a breakpoint at the location in (). You see a little red stop sign appear in line 31.

The next thing to do is to actually *execute* the program, such that you can examine its behavior. Select Program => Run to execute the program; the Run Program dialog appears.



```
(): 'ddd/doc/sample.c:31
```

```
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v
        }
    } while (h != 1);
}

int main(int argc, char
{
    int *a;
    int i;

    a = (int *)malloc((
    for (i = 1; i < argc
        a[i] = atoi(argv

    shell_sort(a, argc);

    for (i = 0; i < argc
        printf("%d ", a
    printf("\n");

    free(a);

    return 0;
}
```

```
8 7 5 4
```

```
8000 7000 5000 1000 4000
```

```
Reading symbols from sample...done.
(gdb) break /usr/users/sts1/zeller/ddd/doc/sample.c:31
Breakpoint 1 at 0x8048666: file /usr/users/sts1/zeller/ddd/doc/sample.c, line
31.
(gdb)
```

```
Breakpoint 1 at 0x8048666: file /usr/users/sts1/zeller/ddd/doc/sample.c, line 31.
```

Running the Program

In Run with Arguments, you can now enter arguments for the sample program. Enter the arguments resulting in erroneous behavior here--that is, 8000 7000 5000 1000 4000. Click on Run to start execution with the arguments you just entered.

GDB now starts sample. Execution stops after a few moments as the breakpoint is reached. This is reported in the debugger console.

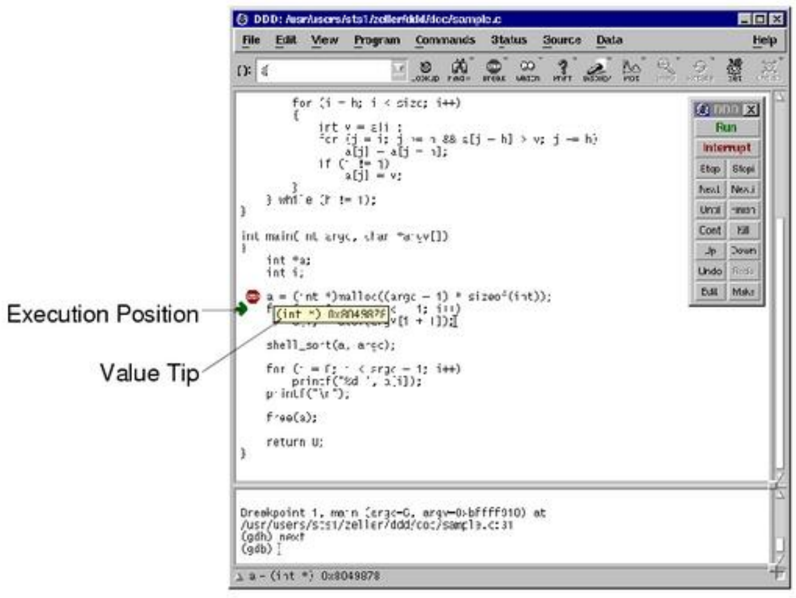
```
(gdb) break sample.c:31
Breakpoint 1 at 0x8048666: file sample.c, line 31.
(gdb) run 8000 7000 5000 1000 4000
Starting program: sample 8000 7000 5000 1000 4000


Breakpoint 1, main (argc=6, argv=0xbffff918) at sample.c:31
(gdb) _
```

The current execution line is indicated by a green arrow.

```
=> a = (int *)malloc((argc - 1) * sizeof(int));
```

You can now examine the variable values. To examine a simple variable, you can simply move the mouse pointer on its name and leave it there. After a second, a small window with the variable value pops up. Try this with argc to see its value (6). The local variable a is not yet initialized; you'll probably see 0x0 or some other invalid pointer value.

To execute the current line, click on the Next button on the command tool. The arrow advances to the following line. Now, point again on a to see that the value has changed and that a has actually been initialized.



```
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

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

    a = (int *)malloc((argc - 1) * sizeof(int));
    f(int *) 0x8049878<  1; i++)
                      v[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}
```

```
Breakpoint 1, main (argc=6, argv=0xbffff910) at
/usr/users/st1/zeller/ddd/doc/sample.c:31
(gdb) next
(gdb)
```

```
a = (int *) 0x8049878
```

Viewing Values in DDD

To examine the individual values of the a array, enter a[0] in the argument field (you can clear it beforehand by clicking on ():) and then click on the Print button. This prints the current value of () in the debugger console. In our case, you'll get

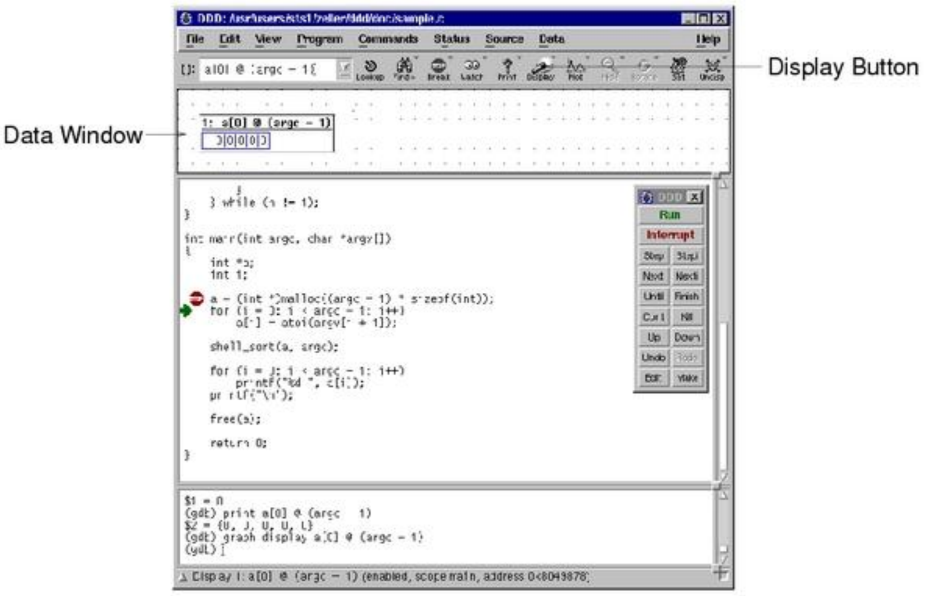
```
(gdb) print a[0]
$1 = 0
(gdb) _
```

or some other value (note that a has only been allocated, but the contents have not yet been initialized).

To see all members of a at once, you must use a special GDB operator. Since a has been allocated dynamically, GDB does not know its size; you must specify it explicitly using the @ operator. Enter a[0]@(argc - 1) in the argument field and click on the Print button. You get the first argc - 1 elements of a, or

```
(gdb) print a[0]@(argc - 1)
$2 = {0, 0, 0, 0, 0}
(gdb) _
```

Rather than using Print at each stop to see the current value of a, you can also *display* a, such that its is automatically displayed. With a[0]@(argc - 1) still being shown in the argument field, click on Display. The contents of a are now shown in a new window, the *Data Window*. Click on Rotate to rotate the array horizontally.


DDD: /usr/users/sts1/zeller/ddd/doc/sample.c
File Edit View Program Commands Status Source Data Help
Lookup Find» Break Watch Print Display Plot Hide Rotate Set Undisp
Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make
Display Button
Data Window

```
(): a[0] @ (argc - 1)

1: a[0] @ (argc - 1)
0|0|0|0|0

    } while (h != 1);
}

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

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);

    return 0;
}

$1 = 0
(gdb) print a[0] @ (argc - 1)
$2 = {0, 0, 0, 0, 0}
(gdb) graph display a[0] @ (argc - 1)
(gdb) 

Display 1: a[0] @ (argc - 1) (enabled, scope main, address 0x8049878)
```

Data Window

Now comes the assignment of a's members:

```
=>  for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);
```

You can now click on Next and Next again to see how the individual members of a are being assigned. Changed members are highlighted.

To resume execution of the loop, use the Until button. This makes GDB execute the program until a line greater than the current is reached. Click on Until until you end at the call of shell_sort in

```
=>  shell_sort(a, argc);
```

At this point, a's contents should be 8000 7000 5000 1000 4000. Click again on Next to step over the call to shell_sort. DDD ends in

```
=>  for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
```

and you see that after shell_sort has finished, the contents of a are 1000, 1913, 4000, 5000, 7000--that is, shell_sort has somehow garbled the contents of a.

To find out what has happened, execute the program once again. This time, you do not skip through the initialization, but jump directly into the shell_sort call. Delete the old breakpoint by selecting it and clicking on Clear. Then, create a new breakpoint in line 35 before the call to shell_sort. To execute the program once again, select Program => Run Again.

Once more, DDD ends up before the call to shell_sort:

```
=>  shell_sort(a, argc);
```

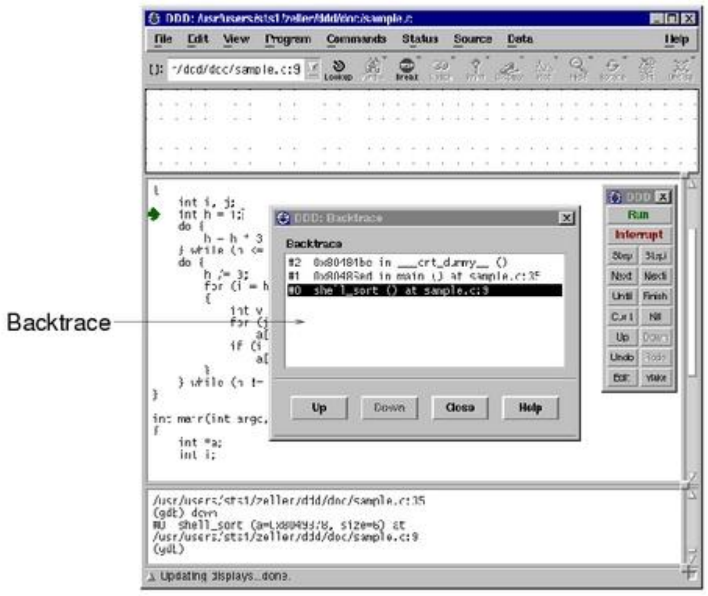
This time, you want to examine closer what shell_sort is doing. Click on Step to step into the call to shell_sort. This leaves your program in the first executable line, or

```
=> int h = 1;
```

while the debugger console tells us the function just entered:

```
(gdb) step
shell_sort (a=0x8049878, size=6) at sample.c:9
(gdb) _
```

This output that shows the function where sample is now suspended (and its arguments) is called a *stack frame display*. It shows a summary of the stack. You can use Status => Backtrace to see where you are in the stack as a whole; selecting a line (or clicking on Up and Down) will let you move through the stack. Note how the a display disappears when its frame is left.



```
(): ~/dcd/doc/sample.c:9
```

```
{
    int i, j;
    int h = 1;
    do {
        h = h * 3
    } while (h <=
    do {
        h /= 3;
        for (i = h
        {
            int v
            for (j
                a[
            if (i
                a[
        }
    } while (h !=
}

int main(int argc,
{
    int *a;
    int i;
```

Backtrace

```
#2  0x80481bc in __crt_dummy__ ()
#1  0x80485ed in main () at sample.c:35
#0  shell_sort () at sample.c:9
```

```
/usr/users/sts1/zeller/ddd/doc/sample.c:35
(gdb) down
#0  shell_sort (a=0x80493:8, size=6) at
/usr/users/sts1/zeller/ddd/doc/sample.c:9
(gdb)
```

```
Updating displays...done.
```

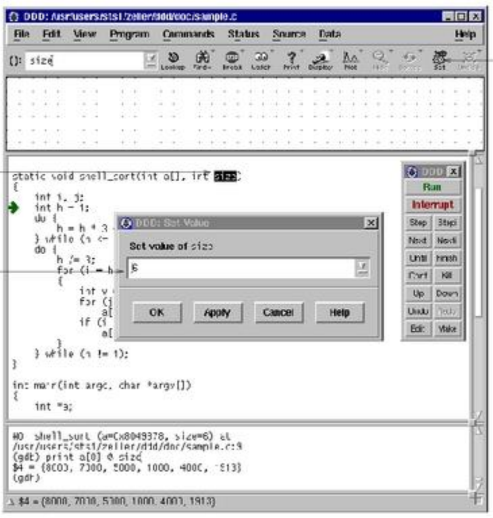
The DDD Backtrace

Let us now check whether shell_sort's arguments are correct. After returning to the lowest frame, enter a[0]@size in the argument field and click on Print:

```
(gdb) print a[0] @ size
$4 = {8000, 7000, 5000, 1000, 4000, 1913}
(gdb) _
```

Surprise! Where does this additional value 1913 come from? The answer is simple: The array size as passed in size to shell_sort is *too large by one*--1913 is a bogus value which happens to reside in memory after a. And this last value is being sorted in as well.

To see whether this is actually the problem cause, you can now assign the correct value to size. Select size in the source code and click on Set. A dialog pops up where you can edit the variable value.


Set Button
Select variable in the source
Edit value
DDD: /usr/users/sts1/zeller/ddd/doc/sample.c
File Edit View Program Commands Status Source Data Help
(): size
Lookup Find» Break Watch Print Display Plot Set
DDD: Set Value
Set value of size
OK Apply Cancel Help
DDD
Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make

```
static void shell_sort(int a[], int size)
{
    int i, j;
    int h = 1;
    do {
        h = h * 3
    } while (h <=
    do {
        h /= 3;
        for (i = h;
        {
            int v
            for (j
                a[
            if (i
                a[
        }
    } while (h != 1);
}

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

```
#0  shell_sort (a=0x8049878, size=6) at
/usr/users/sts1/zeller/ddd/doc/sample.c:9
(gdb) print a[0] @ size
$4 = {8000, 7000, 5000, 1000, 4000, 1913}
(gdb)
```

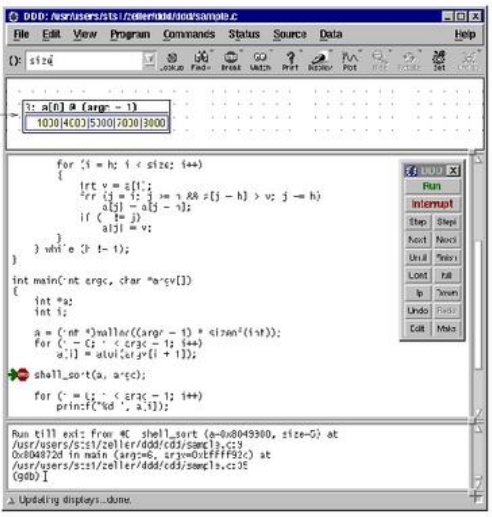
```
$4 = {8000, 7000, 5000, 1000, 4000, 1913}
```

Setting a Value

Change the value of size to 5 and click on OK. Then, click on Finish to resume execution of the shell_sort function:

```
(gdb) set variable size = 5
(gdb) finish
Run till exit from #0  shell_sort (a=0x8049878, size=5) at sample.c:9
0x80486ed in main (argc=6, argv=0xbffff918) at sample.c:35
(gdb) _
```

Success! The a display now contains the correct values 1000, 4000, 5000, 7000, 8000.


Changed values
DDD: /usr/users/sts1/zeller/ddd/ddd/sample.c
File Edit View Program Commands Status Source Data Help
(): size
Lookup Find» Break Watch Print Display Plot Set
DDD
Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make

```
3: a[0] @ (argc - 1)
1000|4000|5000|7000|8000
```

```
        for (i = h; i < size; i++)
        {
            int v = a[i];
            for (j = i; j >= h && a[j - h] > v; j -= h)
                a[j] = a[j - h];
            if (i != j)
                a[j] = v;
        }
    } while (h != 1);
}

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

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    shell_sort(a, argc);

    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
```

```
Run till exit from #0  shell_sort (a=0x8049300, size=5) at
/usr/users/sts1/zeller/ddd/cdd/sample.c:9
0x804872d in main (argc=6, argv=0xbffff92c) at
/usr/users/sts1/zeller/ddd/cdd/sample.c:35
(gdb) I
```

```
Updating displays...done.
```

Changed Values after Setting

You can verify that these values are actually printed to standard output by further executing the program. Click on Cont to continue execution.

```
(gdb) cont
1000 4000 5000 7000 8000

Program exited normally.
(gdb) _
```

The message Program exited normally. is from GDB; it indicates that the sample program has finished executing.
Having found the problem cause, you can now fix the source code. Click on Edit to edit sample.c, and change the line

```
shell_sort(a, argc);
```

to the correct invocation

```
shell_sort(a, argc - 1);
```

You can now recompile sample

```
$ gcc -g -o sample sample.c
$ _
```

and verify (via Program => Run Again) that sample works fine now.

```
(gdb) run
`sample' has changed; re-reading symbols.
Reading in symbols...done.
Starting program: sample 8000 7000 5000 1000 4000
1000 4000 5000 7000 8000
Program exited normally.
(gdb) _
```

All is done; the program works fine now. You can end this DDD session with Program => Exit or Ctrl+Q.

Index