Assembly Programming HOWTO

Download all the required stuff

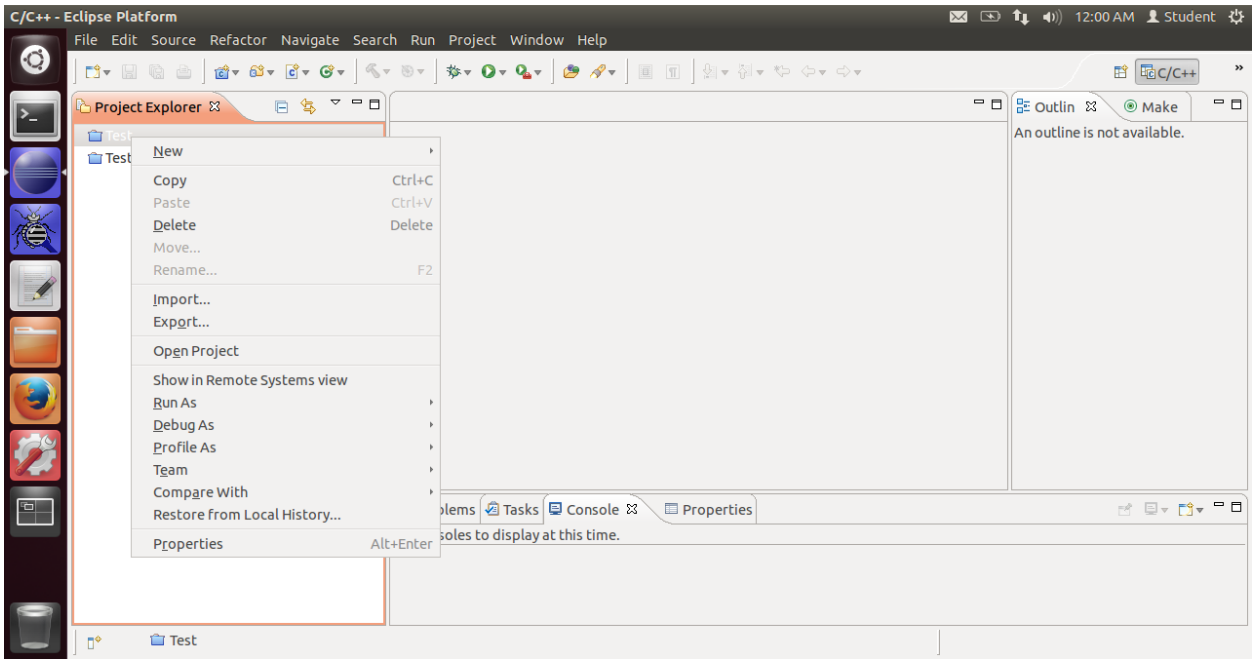
1. Download VirtualBox for Windows or Mac.
2. Download the VirtualBox extension pack.
3. Download the VM for Assembly Programming.

Install VirtualBox and the VM

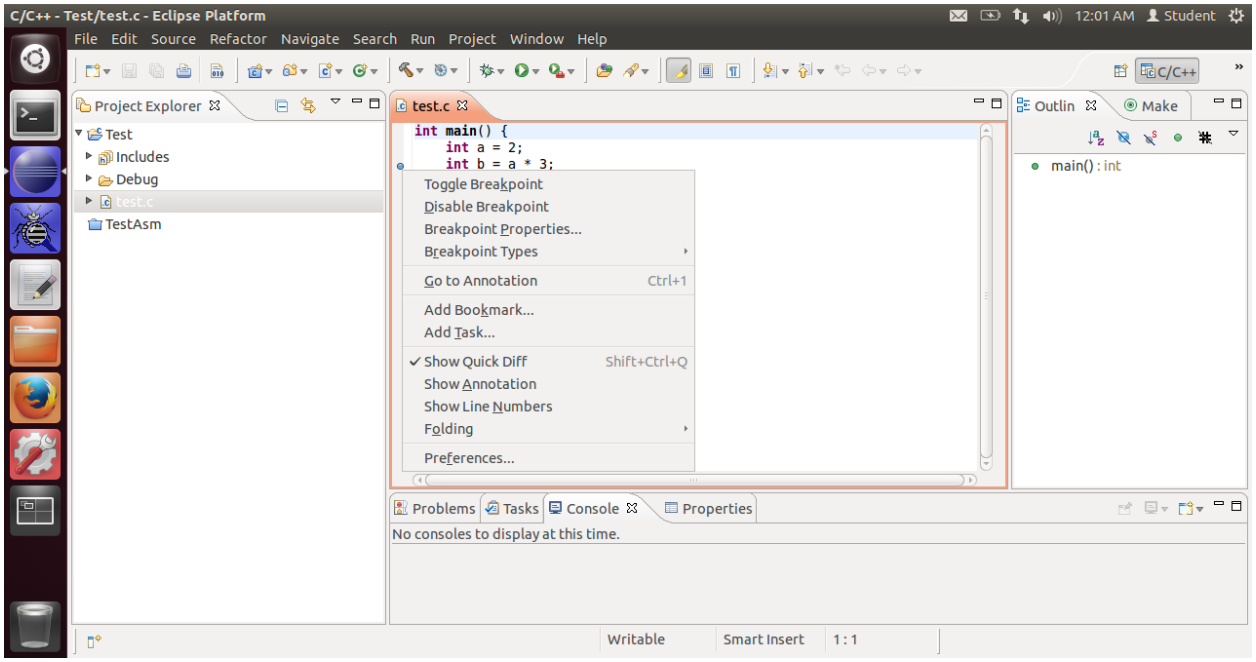
1. To install VirtualBox, just doubleclick on the EXE and follow instructions on the screen. On Mac, double click the DMG file, then drag VirtualBox into Applications.
2. Run VirtualBox from your Start/Applications Menu.
3. Go to File > Settings > Extensions. Click on the Add Package button on the right. Select the extension pack you downloaded earlier. Follow onscreen instructions to complete installation.
4. Go to File > Import Appliance.
5. Point to the .OVA file you downloaded above. This will usually be in your Downloads folder.
6. Once the Import is complete, select AssemblyProgramming from the list of VMs and click on Start.
7. Your VM should now boot. If you get some popups in blue on the top, you can safely close them.

Begin Assembly Programming with Eclipse

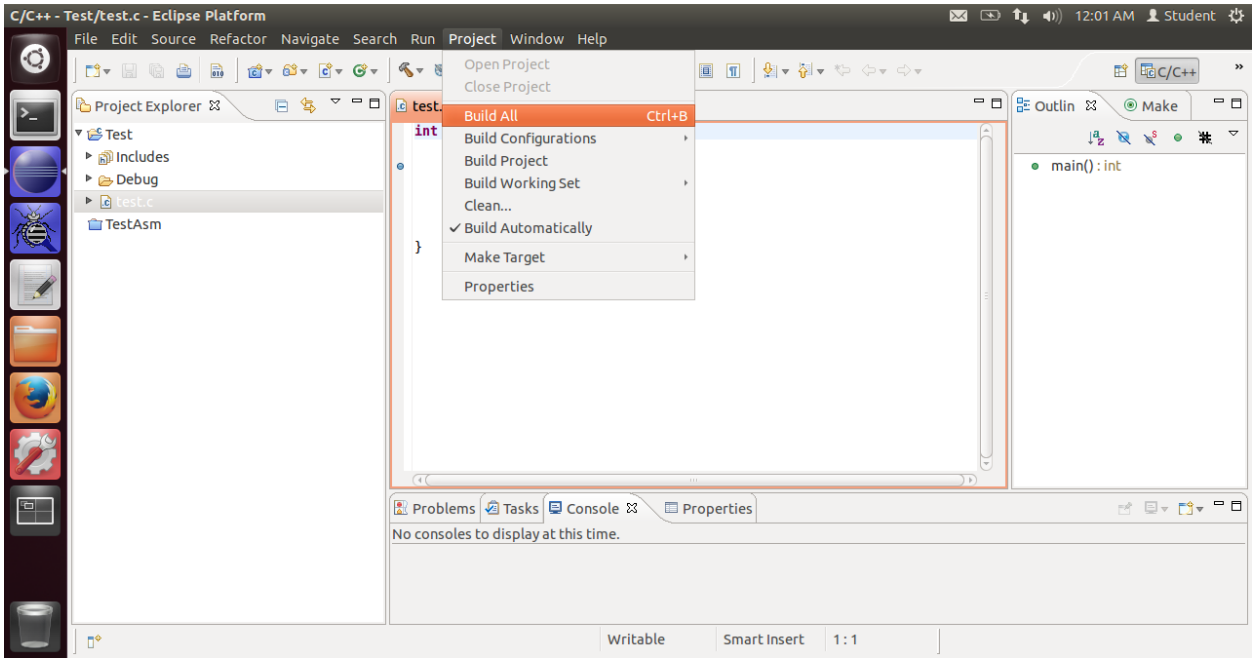
Eclipse should autostart as your VM boots up. Two sample projects exist, one with a C program and one with an ASM program. Open the Test project:



After opening the project, open the associated program file, in this case *test.c*. You can now toggle breakpoints at specific lines of code. Breakpoints pause the program execution so that we may view values of program variables:



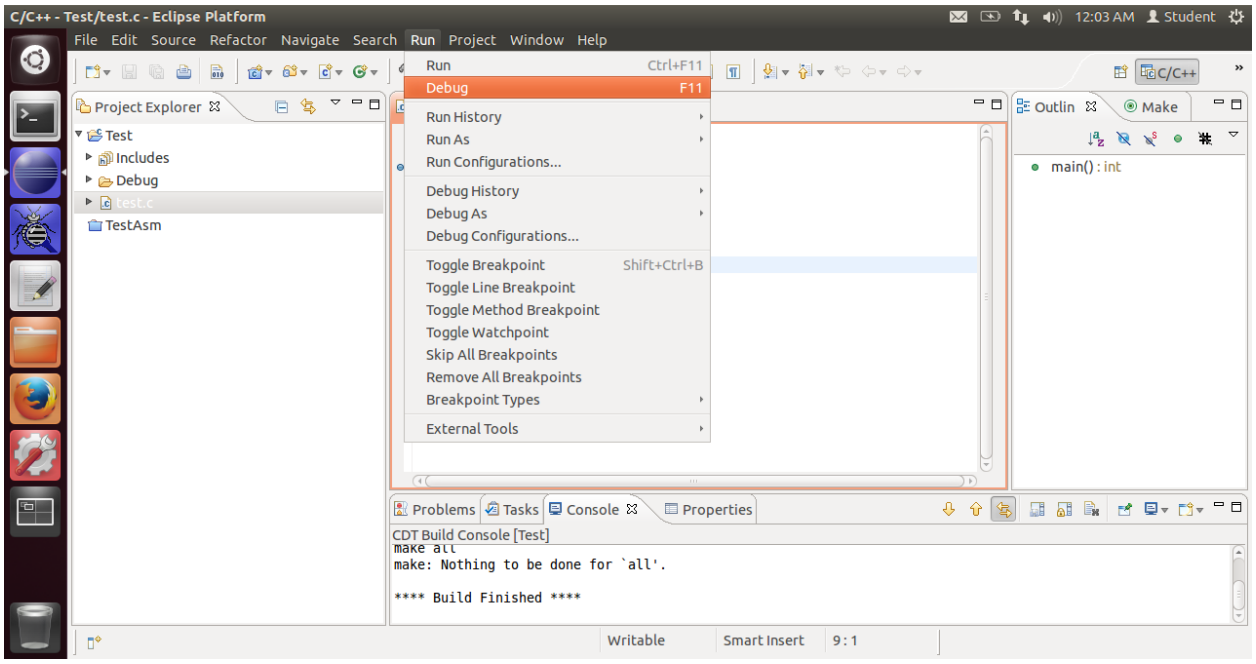
Now we have to Build our code, which is just a fancy name for compiling it. Select `Project > Build All`.



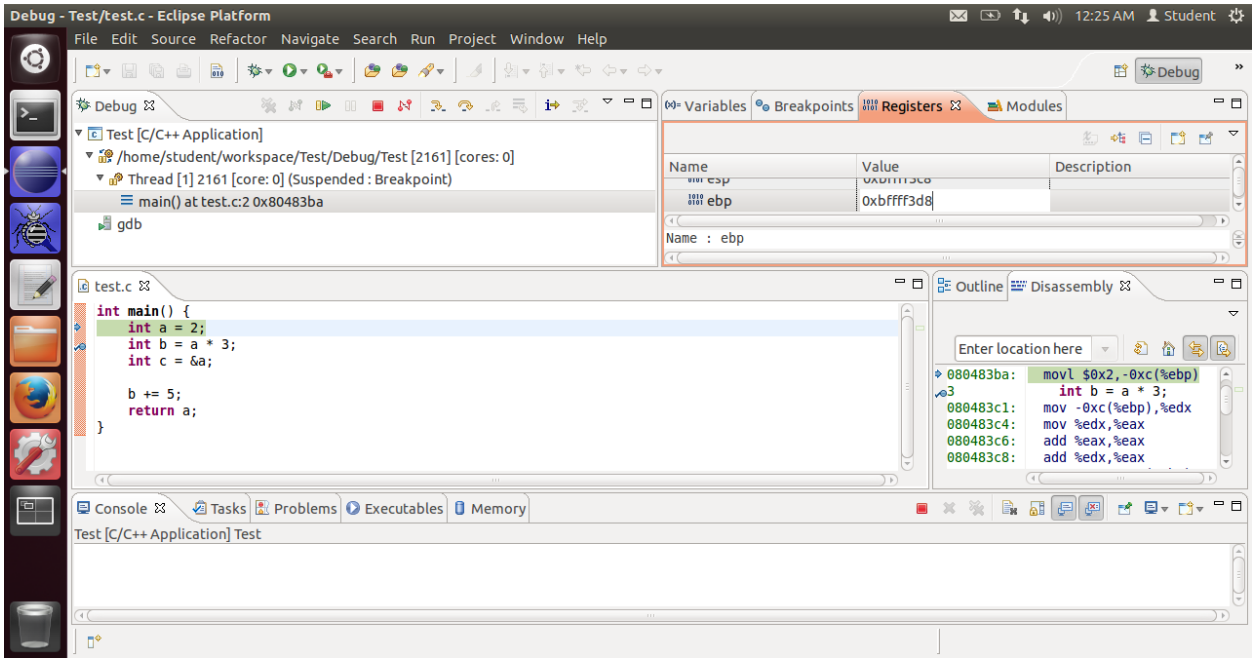
In this case, you are using a sample program provided by me in working condition. Therefore the Build will go through painlessly. Once you write your own programs, you will see a number of errors showing up here. These errors will help you fix your compilation errors.

Just because a program compiles does not mean it is error free. Now we will debug the program to see if it actually does what we intended it to do.

To start debugging, go to `Run > Debug` in the menu.



Eclipse will now change the *perspective* to *debug mode*. You will see a number of windows with all kinds of information about your program.



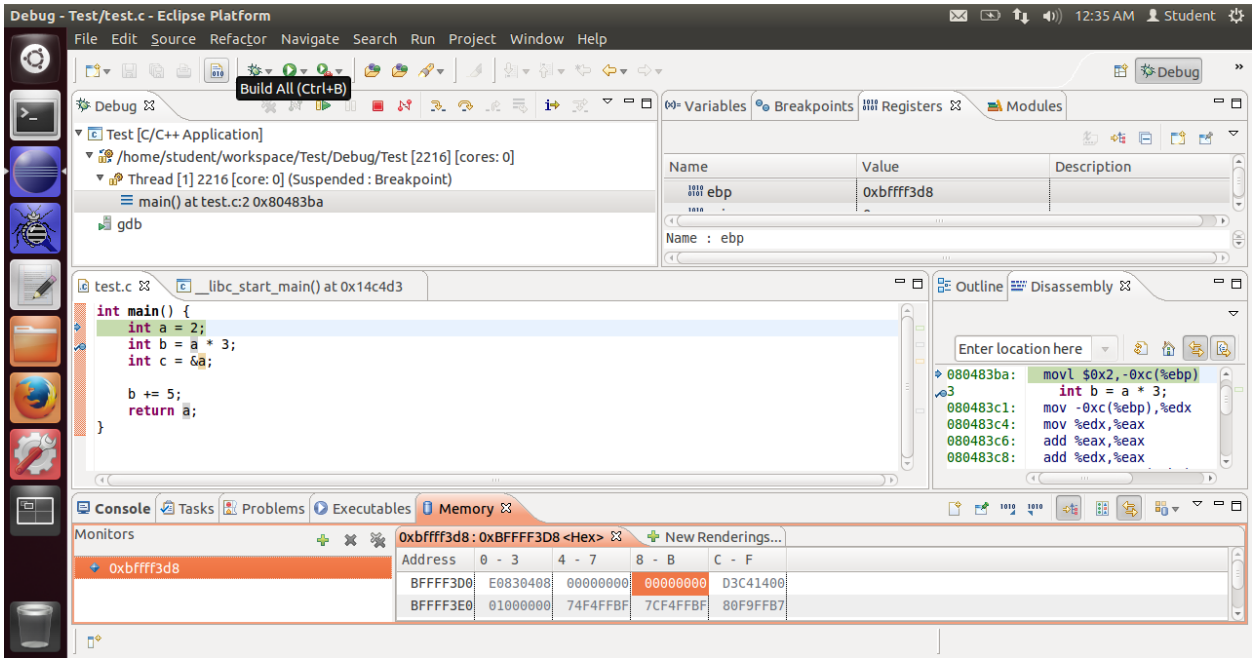
Here is a quick review of all the things you see on this screen:

1. The code file, **test.c**
2. **Disassembly** of the C program with the corresponding assembly statements.
3. **Registers**
4. **Variables**
5. **Memory**. More information on accessing the right memory locations is given below.

It is interesting to observe how a program modifies memory. However, to know where to look in all the memory the computer has, you must first know where your program is storing useful information. This is easy to find using disassembly and registers.

First, note that the first line of C code is (highlighted in light green colour on the left)

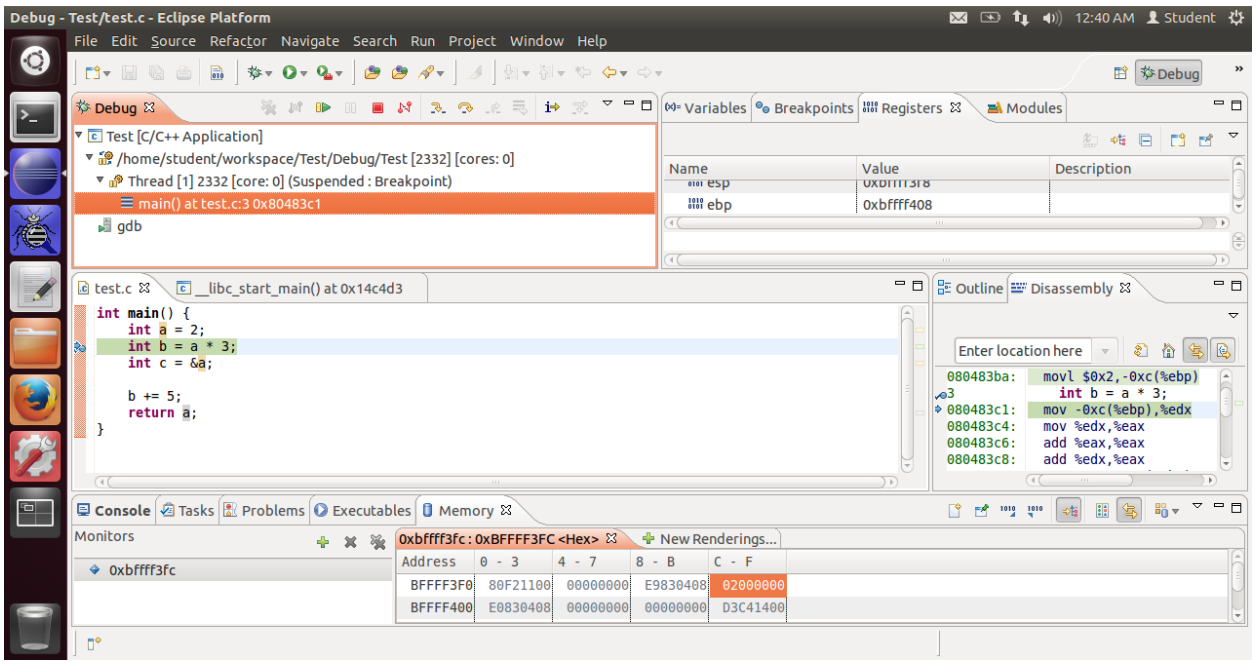
```
int a = 2;
```



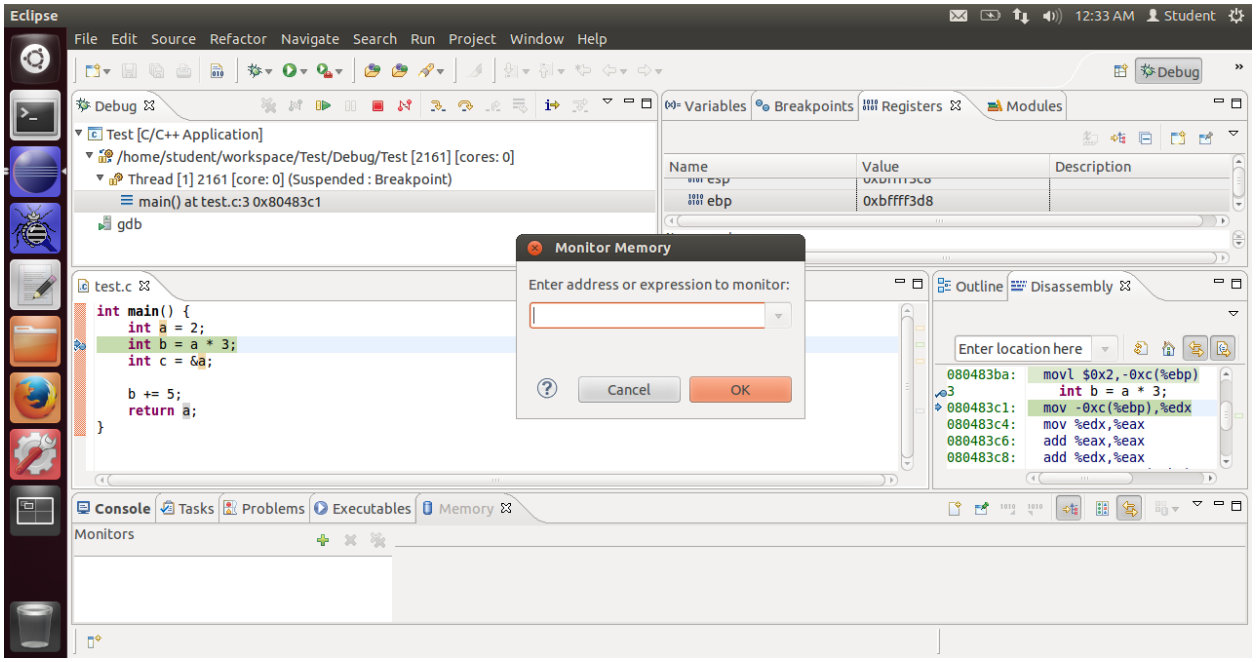
See the disassembly window. Observe the corresponding assembly line (highlighted in light green colour on the right):

```
movl $0x2, -0xc(%ebp)
```

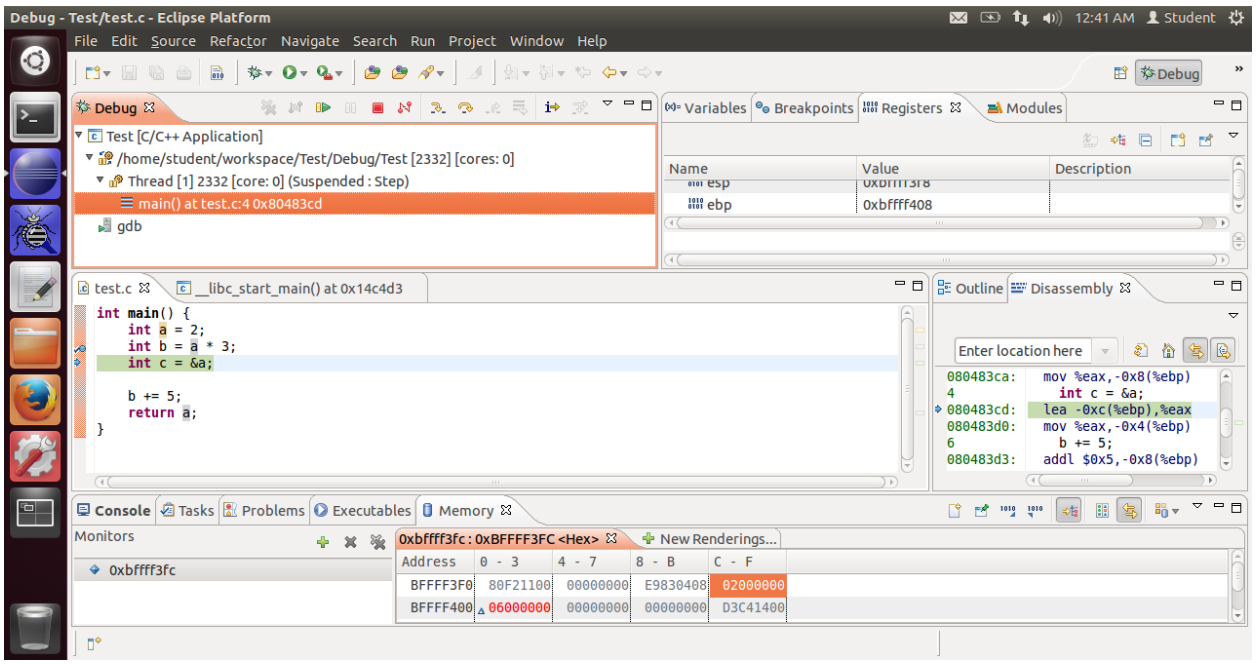
The above line indicates that the program has been compiled such that the variable "`a`" is stored in the stack at (`EBP-0x0c`).



Now, in the Registers tab (see image above), note that the value of `EBP` is `0x3ffff408`. So we have to open the location `0x3ffff408 - 0x0c = 0x3ffff3cc` in memory. In order to do that, first click on the Memory tab, then click on the green plus (+) icon. Now enter the memory address in the dialog box that appears as shown below.



You will now see the contents of different memory addresses starting at `0x3ffff3c0`:



Observe that the contents of the memory address `0x3ffff3cc` is `0x00000002`. This corresponds to the first statement in the C program: `int a = 2;`

Memory addresses are a *word* long. In this particular example (32 bit x86 processor running Linux), each word is four bytes long. Therefore the variable a is stored in addresses `0x3ffff3cc` to `0x3ffff3cf`. Registers such as EBP and EDX are also a word long.

In the image above, you can also see that address `0x3ffff400` is highlighted in red. why is that so? The next line of this code is as follows

```
int b = a * 3;
```

In the Disassembler window, you can see that the corresponding assembly code is:

```
080483c1:   mov -0xc(%ebp),%edx
080483c4:   mov %edx,%eax
080483c6:   add %eax,%eax
080483c8:   add %edx,%eax
080483ca:   mov %eax,-0x8(%ebp)
```

(The values `080483c1` etc. here correspond to the location in memory where the particular assembly instruction exists.)

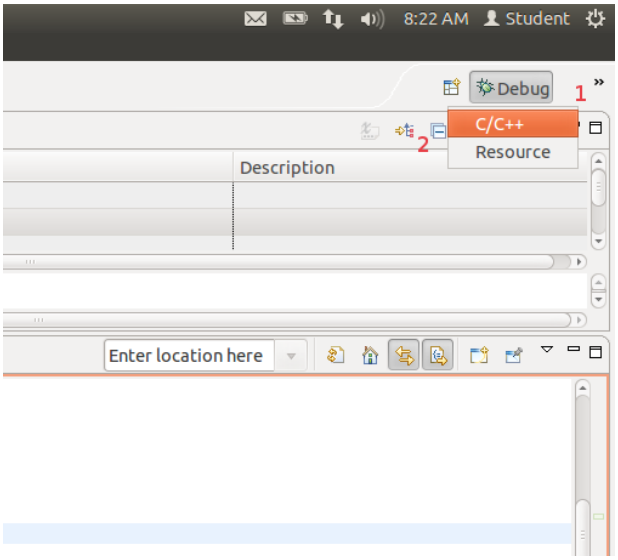
As you can see, first we copy (`mov`) the contents of memory address `EBP-0x0c` to register `EDX`, so `EDX` has the value `2`. We then copy the contents of `EDX` to `EAX`. We then add `EAX` to `EAX`, then `EDX` to `EAX`. Finally, we copy `EAX` to `EBP-0x08`. `EBP-0x08` is where the variable `b` is stored.

We asked to multiply `a` by `3` and store in `b`. Instead, it added `a` thrice and stored it in `b`.

This is known as *compiler optimization*. The compiler (in this case `gcc`, the GNU C Compiler), decided that it would be more efficient for this processor to add it thrice, rather than multiply it by 3 once.

Now, let's revisit the red text. What the debugger is doing is to highlight for us the memory contents that got changed by the instruction that was just executed. But don't programs run all at once? Not in a debugger! Remember we added a breakpoint when we first saw the C program. The program had stopped at that location (after `int a = 2;`). Before I took the screenshot, I selected `Run > Step Into` from the menu first. This caused the debugger to run the next statement (`int b = a * 3;`) , highlight in red what changed, and stop again. Now we can press `F5` (the shortcut key for `Step Into`) and get the next statement to run. On each statement run, certain registers or memory contents should change.

Eclipse supports various "*views*". To go back to code view. Select `Run > Terminate` from the menu. Then, click on the >> button on the top right, then click on `C/C++`.

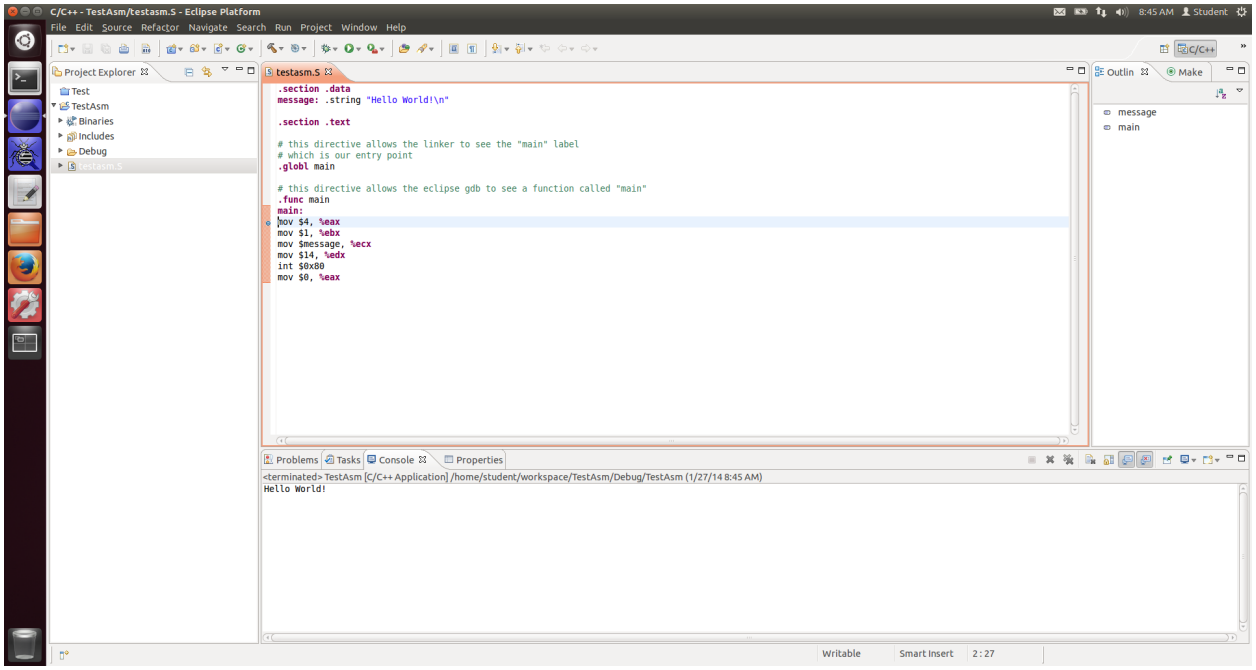


Working with an Actual Assembly Program

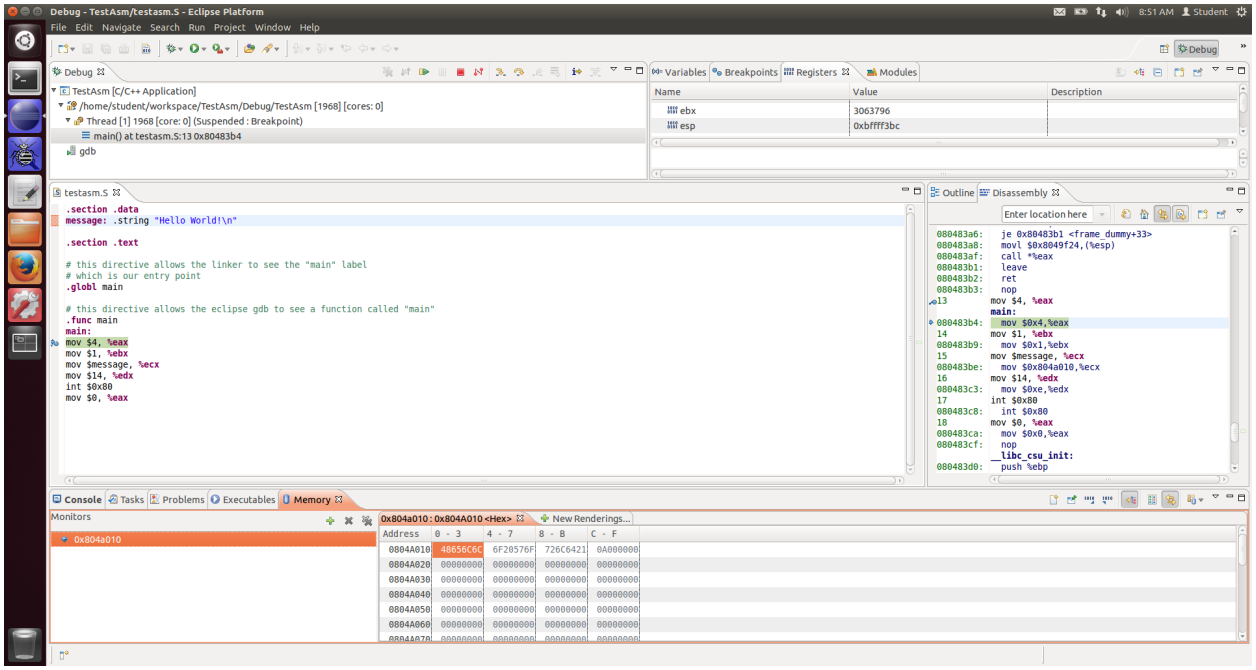
So far, we've looked at the disassembly of a C program. Now let's write an actual assembly program.

First, right click on `Test` in the `Project Explorer` and select `Close Project`. Now, right click on `TestAsm` and select `Open Project`. Double click on the file `testasm.S`.

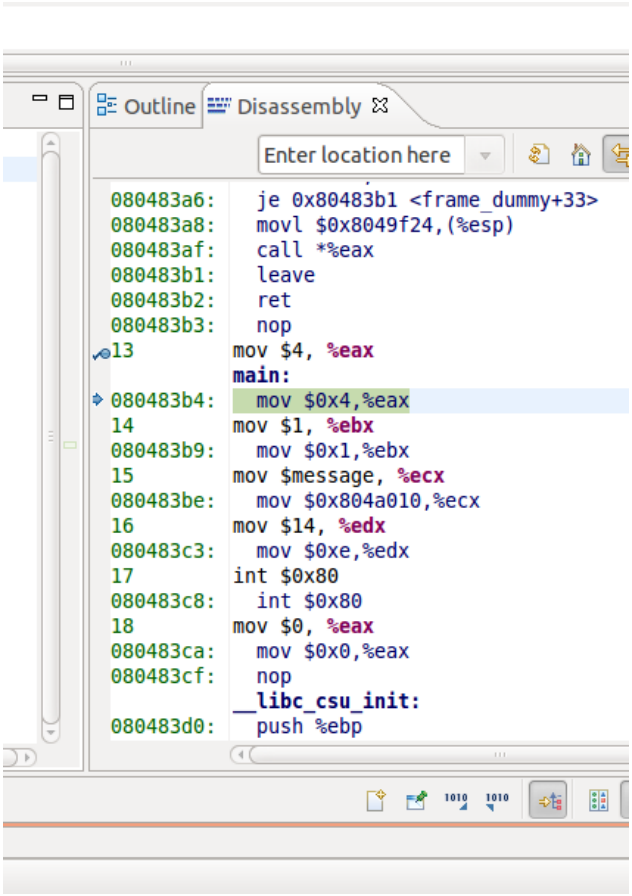
From here on, everything is pretty much the same as before. You will see some assembly code. Go ahead and add a breakpoint. Now `Build All`, and `Run`. You should see something like this:



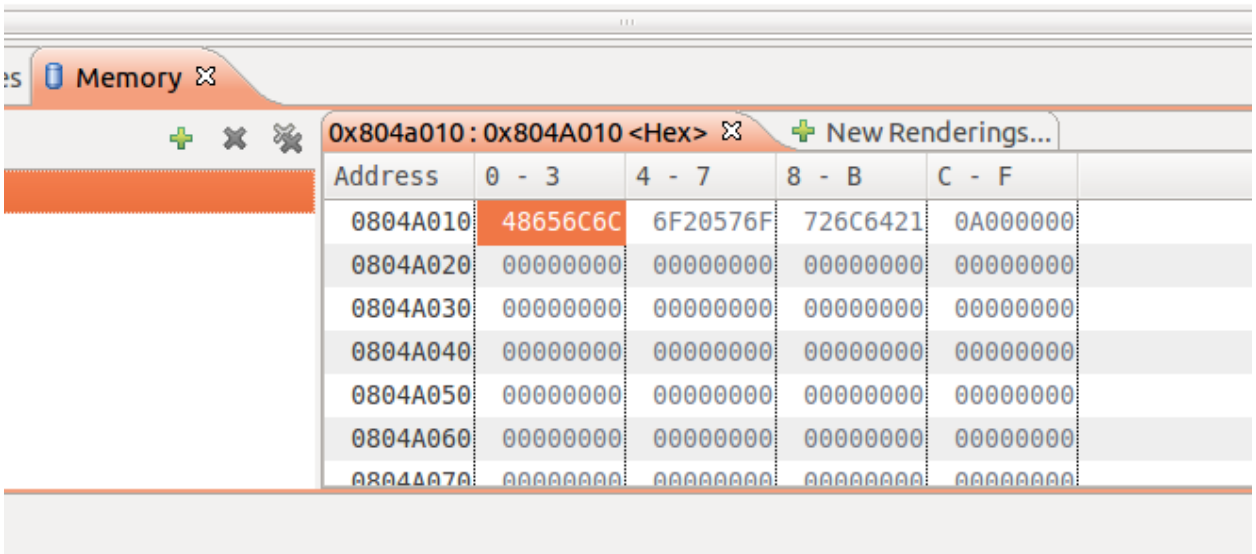
Down below in the console, you can see the output: `Hello World`. Now let's debug this program (although there aren't any real bugs). Go to `Run > Debug`. Your screen should change to something like this:



The disassembly on the right is pretty clean. Let's zoom in on that.



Observe that `$message` has been replaced with `0x804a010`. The variable `message` is actually nothing but an address in memory where the string "`Hello World!\n`" is stored. In C programming we call this a pointer. It is not the value of the pointer that is important… it is what the pointer points to that is of interest to us. So let's go look at this memory address:



Starting with `0x0804A010`, you can see the string "`Hello World!\n`" in all its hexadecimal glory. Let's pull up an ASCII chart here to decode what this is. I'll do the easy ones. `0x0804A015` is `0x20`, which is a space. `0x0804A01B` is `0x21`, which is an exclamation mark. `0x0804A01C` is `0x0A`, which is a new line character.

Our assembly code seems to be more intuitive or clean as compared to the assembly code generated by the C compiler. This is one of the reasons sometimes people choose assembly over high level languages. Observe how the compiler converted a multiplication to three sets of additions. The compiler is a computer program too, and occasionally does not necessarily generate the cleanest or most efficient assembly code (we have discussed in class). Where execution efficiency is of importance, assembly programming is your best bet. Some examples are Operating System code, code that interacts with external hardware in a time sensitive manner (think pacemaker or other medical devices), or scientific formulae that must run zillions of times on a supercomputer (matrix multiplication). A huge amount of mathematical formulae are handcoded in assembly by expert programs, and licensed at very high cost to run on supercomputers that predict the weather or help design new drugs.

Writing your own Assembly Program

Now that you are comfortably running the sample programs I've given you, let's get you started on your own first project. First, close any projects you've got open.

`Go to File > New > C++ Project`

Select `Empty Project, Linux GCC`, and give your project a name. Click on `Finish.`

Go to `Project > Properties > C/C++ Build > Settings` from the menu. Click on `GCC Assembler`. Under `Commandline Patterns`, you will see:

```
${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX} ${OUTPUT}
${INPUTS}
```

Change this to:

```
${COMMAND} ${FLAGS} -g --gstabs ${OUTPUT_FLAG} ${OUTPUT_PREFIX}
${OUTPUT} ${INPUTS}
```

This change asks the assembler to include debugging symbols in the compiler binary. This allows the debugger to step through code.

Now create your first program. Do `File > New > Source File`. Give your program a name, remember to end it with `.S`. That's a capital S, do not put in a `.s` or `.asm`.

Now type in your first program, save, build all, fix your compilation errors, and when the build succeeds, run and debug.

What to do when things don't work?

Sometimes, Eclipse will get stuck. You might end up closing so windows or move them around, and then you won't be able to see your registers any more. Or worse…

Here is how to avoid such problems:

1. Try not to play with Eclipse, yet. It is a finicky bit of software.
2. Remember that you can import the .ova file I've given you, as many times as you wish. So if things break, just start afresh.
3. If you start afresh, you'll loose all the code you've written in the VM. Your options are:
    a. Email the code from the /home/student/workspace directory to yourself using the Firefox browser.
    b. VirtualBox allows you to share a directory between your host Operating System (Windows/Mac) and Linux. Copy all your code to your OS that way.
4. Delete the /home/student/.eclipse and /home/student/eclipse folders in Linux. This will make Eclipse forget most things and start afresh.

If you'd like to email me, it is best done with a screenshot of your VM. Go to Machine menu in VirtualBox, and select Take Screenshort. Email the generated PNG file to me.

How to Share files between the VM and your laptop's host OS

The Internet works within your VM, so you can use the browser to open your email and send and receive files. In case you want to share files between your VM and your laptop's host OS (Windows/Mac), here's how to do that.

First, in your VM, open the terminal. Type the following:

```
student@ubuntu:~$ sudo nano /etc/group
```

The password is **student**.

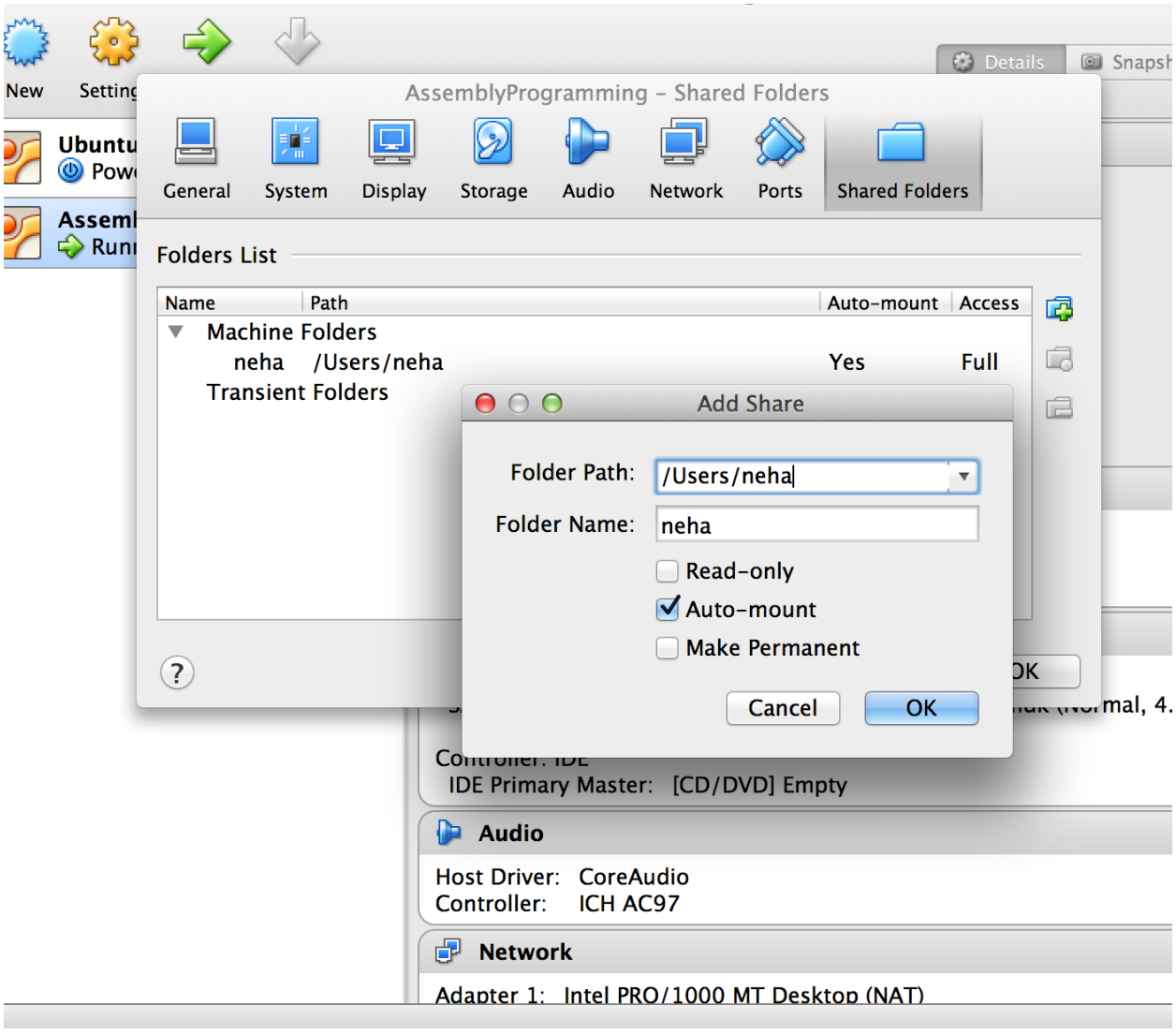
At the end of this file, type the following:

```
vboxsf:x:999:student
```

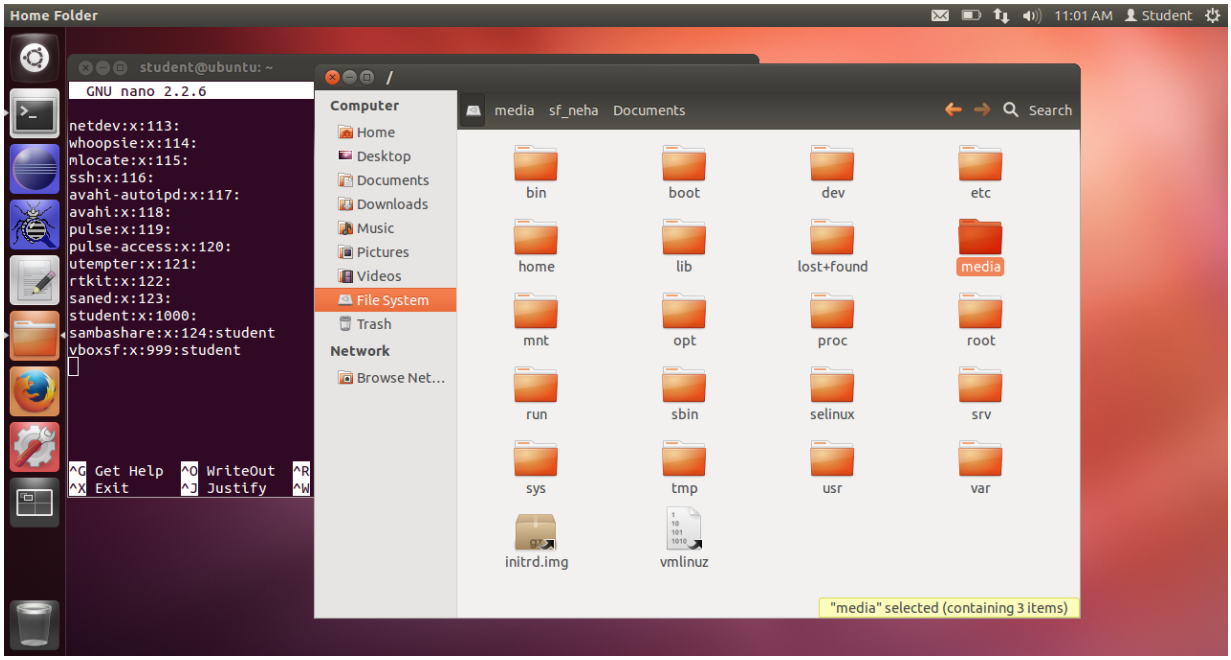
How do you go to the end? Just use the down arrow keys. How do you save and exit? Press Control-o, then Control-x.

Now Shutdown the VM. Click on the gear icon on the top right, and select Shut Down..

Next, in VirtualBox, go to the Settings for this VM. The last option is Shared Folders. In that click on the little folder-plus icon to the right. Select a Folder Path of your preference, and remember to check `Auto Mount` before you hit `OK`.



That's it, you're done. Now Start the VM again. Click on the `folder` icon on the left, then `File System > media`. You should see your shared folder there. Make sure you are able to copy files both from and to this folder.


Home Folder
11:01 AM
Student
student@ubuntu: ~
GNU nano 2.2.6
netdev:x:113:
whoopsie:x:114:
mlocate:x:115:
ssh:x:116:
avahi-autoipd:x:117:
avahi:x:118:
pulse:x:119:
pulse-access:x:120:
utempter:x:121:
rtkit:x:122:
saned:x:123:
student:x:1000:
sambashare:x:124:student
vboxsf:x:999:student
^G Get Help ^O WriteOut ^R
^X Exit ^J Justify ^W
/
Computer
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
File System
Trash
Network
Browse Net...
media
sf_neha
Documents
Search
bin
boot
dev
etc
home
lib
lost+found
media
mnt
opt
proc
root
run
sbin
selinux
srv
sys
tmp
usr
var
initrd.img
vmlinuz
"media" selected (containing 3 items)