

Physics 305: Ch. 1

Getting started with Unix

1.1 An Introduction to Unix

We will be using computers that run under an operating system known as Linux, which is an example of a general class of operating systems known as Unix. Nearly everything you learn about Linux will be true in other versions of Unix.

The Unix operating system and its associated applications is an enormous set of knowledge to come to grips with. It is easy to become frustrated with it. If this is your first encounter, my advice is to start small and learn as you go. You will be able to do many useful things with only a few commands and can learn incrementally as you become more comfortable.

The basic Unix operating system governs how your computer manages its files and runs programs. You will often interact with the operating system through a command-line interface known as a “shell”.

Like Microsoft Windows or the Macintosh operating system, Unix can manage the display of your screen so as to display information in windows or to allow applications to open windows that they control. Unlike Windows or Mac OS, you will tend to manage your files and launch applications from a command line in a text window, rather than by clicking on or dragging icons.

1.1.1 Using the Computers in Steward Room 208

The computers in Steward room 208 are all running a flavor of a UNIX operating system (OS) called Linux. The generic Linux OS is free, but Steward Observatory is using a commercial version called Redhat Linux, which includes many useful packages, tools and user support. You should see the logo “Fedora” on the login screen.

The OS is responsible for managing the file system and peripherals on the computer and allows the user to interact with these systems. A user can execute programs, via the OS, that could perform a calculation, interact with the outside world (via the ethernet), control peripherals (via serial ports) or display something on the monitor

(via the window manager). In UNIX, for example Linux and Mac OS X, the graphical display is simply a program that is being executed by the OS that controls, or manages, what is being displayed on the monitor. There is a clear separation between the OS and the windowing system. The generic windowing system under UNIX is called “X”.

Under Windows (NT, 2000, XP, ect) there is almost no distinction between the OS and the windowing system.

When you sit down at a computer in room 208, there should be a login screen ready to accept your username and password. This means Linux OS is running and the “X” windowing system is also running. If this is not the case, there is something wrong with the computer so move to another. (The computer staff should be informed so the computer can be repaired.) Type in your login name in the field labeled *username* and press return. Then type in your password in the field labeled *password* and press return. This will log you into your account on this computer and start the Gnome window manager. This will look and have a functionality similar to Windows or Mac OS X. With the mouse you can click on icons to run programs, drag files from folder to folder, change settings, etc.

Most of the programming we will be doing in this class will be done directly by the OS rather than through the window manager. Thus, you will need to open an Xterm (X-terminal) which opens a window with a command line interface to the OS. To open an Xterm, press the right mouse button, which brings up a menu, and then move the mouse over X-terminal and left click. You can do this more than once to get multiple terminal windows if you like. It is also possible to create an Icon on the icon bar so that when the icon is clicked a terminal window is started. There are several other icons and menus available on the Linux desktop. Play around with them; there are some interesting and useful tools available.

When you are done using the computer, you must logout of the session (not just the terminal windows!). For the computers in room 208, click the “Redhat” on the icon bar and select “Log Out”. Confirm the logout command in the pop-up window. Don’t forget to logout!

You can log in to sagan.as.arizona.edu from elsewhere on campus, but you will probably want to run an X session. Running an X session on a remote computer will be discussed in a later lecture.

1.1.2 Using the Computers in PAS Room 272

Room 272 in the Physics and Atmospheric Sciences (PAS) building contains about twenty PC’s running Windows. This is where the Physics 305 lectures will be, and the room is reserved for Physics 305 at those times. At other times you may work there, along with other physics students doing work for other courses.

First, a word about using this equipment. The physics department provides and maintains these facilities, and if you wish you should be able to do all of your Physics

305 work there. But some of you will wish to use your home computers, or perhaps computers in some other department. This is fine, and we will even do our best to advise you on problems you may encounter. But there is one big **WARNING**: If the power in the PAS building goes down on the night before an assignment is due, or some other serious problem with the equipment occurs, we will make reasonable accommodations so that you can get your work done. If your home computer goes up in smoke and you lose your work, that is your **tough luck**. So, do this at your own risk. Personally, your instructor regularly works on his home computer, and has developed the habit of copying his work to the Physics department computers at the end of every work session.

To get into room PAS 272, swipe your Catcard in the reader by the door. Hopefully everyone enrolled in this section will have access on the first day. One of the computer staff will be in class the first day to help deal with any problems.

Once inside, choose a workstation. If the person ahead of you remembered to log out, it will be displaying a startup screen. Do what it says — press control-alternate-delete (hold ctrl and alt, and press delete). The computer will then try to put itself into a sensible state. Wait, wait, wait

Then you get a login screen. Your login name is your UA Netid. If you don't have one, you will need to get one. Your password for this computer is your standard UA password.

The actual computing for this class will be done on a machine called faraday, which runs the Linux operating system. We will access this machine using a program called "Secure Shell", or "SSH" for short. There is an icon to start this on your screen. When you start up SSH, choose "connect" at the top. For the machine name put "faraday". Your initial password will be given to you on the first day of class. Once you log in to faraday, change it immediately. Do this by typing "passwd". Then it will ask for your old password, and then your new password (twice).

In addition to SSH, the workstations in PAS 272 run software called an "X server". This interprets graphics commands generated by faraday, allowing you to display plots and other nice things. In particular, we are trying out a new one called "Xming". It is working fine in rehearsal; let us know if you have trouble with it. A quick way to see if it is working is to log in to faraday and type the command "xclock". If this produces a clock on your display, things are working. If Xming should cause trouble, we will first try restarting it (from the programs menu in Windows) and if that fails, try starting an alternate server called "Xwin32".

One final thing — please remember to log out from Windows when you are finished working.

1.1.3 Files and Directories

In Unix, like in Windows or Mac OS, you store information in files. Files might contain text that you've typed, the output of programs you've run, or the machine

instructions of the programs themselves. You can (and should) organize your files into a hierarchy of directories (also known as folders in Mac OS).

Unix allows multiple users to exist on a machine, and therefore each user is given their own private area, known as their “home directory”. All your files and directories will reside in or below your home directory.

Here, then, are some basic commands about files and directories that you’ll need to know.

ls This lists the names of the files in the current directory. Any directories hanging off of the current directory will also have their names shown. The mnemonic is “list”.

mkdir <directory_name> This creates a directory named <directory_name> within the current directory. Important: the notation <variable> is my way of indicating that you should fill in the variable with a name of your choice. You must not include the < or > characters, because these have special meaning in the shell. The mnemonic is “make directory”.

cd <directory_name> This changes your current directory to be the named directory. Any files you now reference will be searched for in the new directory. The mnemonic is “change directory”.

cd .. This changes your current directory to be the parent directory of the current one. In other words, it moves you one level up in the hierarchy.

cd If you omit the directory name entirely, the **cd** command moves you back to your home directory.

pwd This displays the name of your current directory on the screen, in case you forget where you are in your directory structure. The mnemonic is “print working directory”.

rmdir <directory_name> If a directory <directory_name> is empty, this command deletes it. The mnemonic is “remove directory”.

rm <file_name> This deletes the file named <file_name>. It will be gone forever; there is no equivalent of the Trash bin in Unix! The computer will ask you whether you are sure before deleting this file; you can answer **y** for yes if you want to delete the file¹.

¹This interactive mode is turned on by the **-i** option. Without this option, the computer simply deletes the named files, no questions asked! Because this is so dangerous for beginners (and experts), I have customized your shell so that the **-i** option is the default. I tell you this because you may someday use a Unix system for which **-i** is not the default!

`mv <old_file_name> <new_file_name>` This renames a file. Note that if there is already a file with the name `<new_file_name>`, it will be overwritten and gone forever! As above, the computer will ask before overwriting a file¹. The mnemonic is “move”.

`mv <file_name> <directory_name>` This moves the file into the directory, retaining the same file name. If there was already a file of that name in the directory, it will be overwritten after you have been asked if that is ok¹.

`cp <old_file_name> <new_file_name>` This copies the file `<old_file_name>`, creating an exact duplicate with the new name `<new_file_name>`. Again, if there is already a file with the name `<new_file_name>`, it will be overwritten and its contents will be gone forever! The computer will ask you before doing this¹. The mnemonic is “copy”.

`less <file_name>` This prints the contents of the file to your terminal screen, one page at a time. Type the space bar to see the next page, type `b` to go back one page, type `q` to quit the program. The command `less` is actually an improved version of the command `more`, which you may have used in earlier Unix encounters.

Next, there are a number of general commands that one should know:

`passwd` This changes your login password. You’ll be asked to type your old password and then your new one twice. Your password should be at least six characters long and should not be a word in the dictionary. If you forget your password, contact me or the system administrator.

`lpr <file_name>` This will send the named file to the printer. You should only do this with text files and with Postscript files! In particular, if you have a file with an image (like a .GIF or .JPG), you cannot print the image with `lpr`. Instead, you will send the string of raw binary data to the printer, wasting paper! Try using Netscape if you need to print an image.

`ghostview <file_name> &` This creates a window to display a Postscript file. It also will display PDF files. The ampersand (`&`) causes the `ghostview` program to run independently from the shell, so that you can continue to use the command line. Try it without the ampersand to see what the difference is.

`emacs <file_name> &` This starts the editor `emacs` so that you can alter the contents of the named file.

`man <command_name>` This will probably not be useful at first, but this command summons a help page about the named command. Help sounds great, but the so-called “man pages” tend to be exhaustively complete, to the point where

beginners will often feel overwhelmed. Most Unix commands have a huge list of options that turn on and off different features, and the man pages will explain all of them. Still, if you stick with Unix, you will learn to use these pages. The mnemonic is “manual”.

`cc <file_name>` This starts the C compiler to try to named file into an executable program. Much more on this later!

1.1.4 Unix command-line syntax

When you type at the command line of the shell, the first word of each line is special, because it specifies the command to be executed. For now, those commands are simply programs provided by the operating system, but in this course, you will learn to write your own commands.

1.1.5 Options

Unix commands often have many options that you can invoke. The convention obeyed by most commands is that options are designated by following the command with one or more words that begin with hyphens. For example, following the `ls` command with the `-l` option, i.e. typing `ls -l`, will produce a listing of your files with much more information, including the sizes in bytes and the last time you modified the file. Typing `ls -F` will produce the usual short listing, but will append a `/` to all the directories and an asterisk (`*`) to any executable programs. Typing `ls -lF` will combine both options; that’s the same as typing `ls -l -F`.

You can get a list of the options available to each command by using the `man` command described above. You don’t need to learn all of the options, but you should learn to recognize that words on the command line that begin with hyphens are generally there to modify the behavior of the program being executed.

1.1.6 Special keys

There are a number of keys that have special behavior when typing at the command line.

CTRL-c In many Unix programs, holding the control key down while hitting the ‘c’ key will interrupt a program and return you to the command line. If you’ve done something that you want to stop, typing **CTRL-c** will usually stop it.

CTRL-z Again, this means holding down the Control key while hitting the ‘z’ key. This *temporarily* halts a program and returns you to the command line. However, the program is not killed; it is just suspended. You can re-start the program at the point it left off by typing `fg` at the command line (`fg` stands for foreground, and yes there is a background that I’ll tell you about later).

CTRL-s This stops new output from being printed to the screen. Everything will halt while you can read what is already there. However, this state also looks as if the computer is frozen or crashed, which is confusing if you've hit **CTRL-s** by accident! Type **CTRL-q** to start the output again.

Arrow keys The left and right arrows will move the cursor through the text you have already typed. Typing new characters will insert them into the existing line. The up and down arrows will move you through the most recent commands that you've executed! This means that you don't need to re-type commands that you want to repeat. Also, if you mis-typed a command, you can use up-arrow to retrieve the command and then the left-arrow to go back into the command for editing. Hitting `<return>` will execute the command.

TAB If you've typed part of a file name, hitting **TAB** will cause the shell to search the current directory to find any files that match the partial name you've given. The shell will then fill in the rest of the name. In other words, the computer will do some of your typing for you. Try it!

1.1.7 File and directory names

Let's imagine that you have a directory named **Colleges** in your home directory and that you have files named **arizona** and **asu** in that directory. If your current directory is your home directory, but you want to look at the contents of the file **arizona**, you don't have to change directories to **Colleges**. Instead, you can refer to the file by the name **Colleges/arizona**. The slash tells the shell that first piece is a directory and the second piece is the file name. Indeed, if you have a deep hierarchy of directories, you can keep layering on the slashes to refer to files at the bottom of the structure.

Note, however, that if you do enter the **Colleges** directory (with `cd Colleges`), you can no longer refer to the file as **Colleges/arizona**. This would tell the computer to search for a file in the **Colleges** sub-directory of your current directory! For this reason, these kinds of references are called *relative paths*; they depend on where you are.

If you want to refer to a file in such a way that it doesn't matter what your current directory is, you need to use an *absolute path name*. For now, there is only one such syntax that you need to know: the tilde character (`~`) stands for your home directory. Hence, in the above example, the reference `~/Colleges/arizona` would specify the file **arizona** regardless of your current directory.

Two other useful abbreviations is that a single period (`.`) denotes the current directory while two periods (`..`) denotes the parent directory to the current one. Hence, if you are in the **Colleges** directory, typing `mv asu ..` will move the file **asu** to the next level up in the directory hierarchy, in this case your home directory. Alternatively, if you are in your home directory, `mv ~/Colleges/asu .` will again move the **asu** file from the **Colleges** directory to your home directory.

Note that file names in Unix are case-sensitive: the files `Arizona` and `arizona` are different. Finally, note that there is no need for Unix file names to have suffixes as in DOS. In other words, we do not need to call our file `arizona.txt`. However, we will encounter suffix conventions that will be convenient.

1.1.8 Wildcards

A final useful aspect of the command line is that the asterisk (*) will match any file names that match the pattern. For example, if our directory contains the files `arizona.c`, `arizona.o`, and `ucla.c`, we could refer to the first two files by `a*`. For example, `ls a*` will display only these two files, `ls *.c` will show `arizona.c` and `ucla.c`, while a single asterisk (`ls *`) will show all files.

Please note that this can be dangerous! For example, the command `mv *.c` will be expanded to `mv arizona.c ucla.c`, which means that the file `arizona.c` will be renamed as `ucla.c`, replacing the contents of that file. Similarly, `rm *` will delete all files in your current directory!

Hence, when beginning, use the asterisk wildcard sparingly, if at all! If you are not sure what a particular pattern will expand into, try it with `ls`.

1.1.9 Web-based tutorials

The web has many Unix tutorials. A few are listed on the course home page <http://cmb.as.arizona.edu/eisenste/phys305/>.

- <http://snap.nlc.dcccd.edu/learn/idaho/unixindex.html>
- <http://www.ee.surrey.ac.uk/Teaching/Unix/>
- <http://www.cs.bu.edu/teaching>

1.2 An Introduction to Emacs

In this course, you will need a way to edit text files. You are welcome to use any editor you like. In particular, if you're already familiar with a Unix editor such as `vi`, feel free to keep using it; we will describe `vi` in the next section. Note that an editor is slightly different than a word processor like Microsoft Word; text files in Unix contain no internal formatting (like boldface text or justified margins) and the job of the editor is only to edit the contents of the file not to adjust the appearance.

If you are new to Unix and need to learn an editor, I would recommend learning `emacs`. `Emacs` is a very big and powerful program, however, so you will need to start with the basics and grow from there.

When you type `emacs <file_name> &` at the command line, a window will appear in which you can see the contents of the named file and type or delete as you see fit. The arrow keys can be used to move the location of the cursor, or you can use the left-button of the mouse to re-position the cursor. To save what you have done, hold the control key down and hit `x` and then `s`. This is usually written as `Ctrl-x Ctrl-s`. To quit `emacs`, type `Ctrl-x Ctrl-c`.

Many commands in `emacs` are accessed by using the Control key in this fashion. For example, `Ctrl-a` will move to the beginning of the current line, while `Ctrl-e` will move to the end of that line. Other commands are accessed by the "Meta" key; for example, `Meta-a` moves to the beginning of a sentence. However, there is no key called "Meta" on your keyboard! Instead, it is called "Alt". Also, you can access Meta commands by the "Esc" key; unlike "Alt", you simply type "Esc" and then the next key (e.g. `a` in the `Meta-a` example) rather than holding it down. Try it!

If you find yourself in a situation where `emacs` is prompting you for input that you don't want to give (like the subject of a Find command), typing `Ctrl-g` will cancel the previous keystrokes and return you to the initial environment.

Modern implementations of `emacs` have a set of pull-down menus along the top of the window that will give you access to many of the popular commands. All of these commands have a keystroke equivalent, but you will probably find it easier to start with the menus.

When you save a file with `emacs`, `emacs` keeps the previous version for you. If your file is called `arizona.c`, the old version is in `arizona.c~`. Only the next-to-last version is saved, so if you really want to keep a version for posterity, you should make a copy with a new name. An example of this is if you have a working version of a C program and you want to experiment with a new way to do a step!

If you are logging in to `sagan` from a Windows or Mac elsewhere on campus, you may be limited to a simple text window. In this case, `emacs` is not able to create a new window on your monitor. You can run `emacs` directly in your text window by adding the `-nw` option, i.e. `emacs -nw <file_name>` (with no ampersand). However, `emacs` will now lack menus, so you must give the commands simply by keystroke.

If you are logging in to `sagan` from another Unix machine or a computer running

the X environment and if you log in to sagan using `ssh sagan` (as opposed to `telnet`), then `emacs` should be able to create windows as usual.

A good introduction to `emacs` is given by its online tutorial. You can start this either by selecting “Emacs Tutorial” from the “Help” menu or by typing `Ctrl-h t`. If you intend to learn and use `emacs`, you should study this tutorial!

You can find additional `emacs` tutorials on the web. The course web page has a few listed:

- <http://www2.lib.uchicago.edu/~keith//tcl-course/emacs-tutorial.html>
- <http://xahlee.org/emacs/emacs.html>
- <http://www.linuxhelp.net/guides/emacs/>

Here are some quick reference pages, just to list the commands.

- <http://www.cs.rutgers.edu/LCSR-Computing/some-docs/emacs-chart.html>
- webdev.apl.jhu.edu/~rbe/java/.../Emacs_Quick_Reference.pdf

Good luck!

1.3 The vi text editor

If you instead want to use the vi editor, that's fine too. Here are some notes to get you started.

The first thing you have to know about text editors is that the keyboard has to serve two purposes. Sometimes when you type something it is text that is to be entered into the file you are editing, and sometimes it is a command to do something to the file. For example, typing ‘‘delete line 15’’ might mean to enter the text “delete line 15” into the file (I just did that in making these notes), or it might mean that whatever is on line 15 of your file is no longer wanted. It isn't really practical to have two keyboards, so a text editor must have some way to distinguish the two kinds of input. In the vi editor this is done by having two modes, the “command mode” and “insert mode”. They are just what they sound like. In the command mode, you are telling the computer to do something to the file, and in the “insert mode” you are entering text into the file. When you start the editor, you are in the command mode. One (of many) ways to get into the insert mode is to hit “i”. Then, what you type will be placed in the file, beginning at the location of the cursor. To get out of the insert mode and back into the command mode, hit the escape key, usually abbreviated *esc*.

Every frustrated beginner sooner or later ends up trying to type commands, and finding that everything just appears on the screen as you type it. This is probably because you are in the insert mode. The escape key is your friend. When you don't understand what is happening, try hitting the escape key!

You will spend a lot of time editing files, so it is a good thing that the common commands have very short abbreviations. For example, you don't really type “delete three lines”, instead you type “3dd”. This is annoying at first, since the abbreviations are not obvious and you will always be referring to your references. But after a day or so you will be doing things quickly, if not accurately.

Here is a really minimal set of commands, just to get you started. This is mostly just to convince you that this really isn't that complicated. But you will soon want to know more commands so that you can do things more efficiently.

There are really only five things you need to know how to do: start the editor, stop the editor, move around in the file, add something and remove something.

1. To start the editor: “vi *filename*”. Here *filename* is the name of the file you want to edit. It can be a file that already exists that you want to change, or a new file that you want to create.
2. To stop the editor: “:x”. “x” stands for “exit”. Watch out for that colon! This writes the file you are editing to disk, and stops the editor.
3. To move around: Lots of ways, but start with the arrow keys: ↓, ↑, → and ←.

4. To add something: “i”, for “insert”, then type whatever you want to put in. Then hit *esc* to stop inserting.
5. To remove something: Lots of ways. start with “x”, which deletes the character at the cursor.

Now go try it. After you have edited a couple of files, continue to the next page where there are more commands that will let you do your work faster. They are all variations on the same five functions.

Actually, before proceeding, you may find that in your drunken stupor you have irretrievably mangled the file you were trying to fix. If this happens, and you want to get out of the editor without saving all the changes you have made, type “:q!”. Again, note the colon. Here “q” stands for “quit”, and the exclamation point tells the computer that you really do mean to quit without saving your changes.

Now here is a bigger table of commands. It still isn’t all of them, but it is probably good enough.

- **Starting the editor**

Edit one file *vi filename*
 Edit several files *vi file1 file 2 ...* (“:n” moves to next file)

- **Stopping the editor**

Save changes and quit :x
 Quit if you haven’t made changes :q
 Discard changes and quit :q!

- **Moving around**

Up	“↑” or “k”	
Down	“↓” or “j”	
Right	“→” or “l”	
Left	“←” or “h”	(because “h,j,k,l” are right under your fingers.)
End of line	“\$”	
Beginning of next line	<i>return</i>	
Beginning of previous line	“_”	
Down 1/2 page	ctrl-d	(hold ctrl key, hit “d”)
Up 1/2 page	ctrl-u	(“up”)
Down full page	ctrl-f	(“forward”)
Up full page	ctrl-b	(“backward”)
Line number 77	“:77”	etc.
Last line	“:\$”	(“\$” means “end”)
Next occurrence of “pattern”	“/pattern”	<i>ret</i>
Previous occurrence of “pattern”	“?pattern”	<i>ret</i>
Repeat last search	“n”	(stands for “next”)

- **Adding stuff** (“esc” to end additions)
 - Add before cursor i (“insert”)
 - Add after cursor a (“append”)
 - Start new line below cursor o (“open”)
- **Removing stuff**
 - Delete one character x
 - Delete one word dw
 - Delete one Word dW includes punctuation
 - Delete one line dd
 - Delete four lines 4dd etc.
 - Delete three words 3dw etc.
- **Copy and paste**
 - Grab one line Y (“yank”)
 - Grab five lines 5Y etc.
 - Paste last thing grabbed p (“put”)
- **Miscellaneous**
 - Undo the last command u **USEFUL!**
 - Repeat last modification . **USEFUL**
 - Built-in help :help (might help)

There is an online manual for the “improved” vi editor at <http://vimdoc.sourceforge.net>, and probably other places. There are several books on vi. One good one is the O’Reilly Book *Learning the vi editor*, by Lamb and Robbins. (The “O’Reilly books” are the ones with cute pictures of animals on the cover, and are generally very good.)

1.4 Turning in Homework

Homework for Physics 305 will be handed in electronically by email. The account for section 1 (Toussant) is p305@physics.arizona.edu. The account for section 2 (Eisenstein) is miao@physics.arizona.edu.

Although the computers in SO 208 and PAS 272 do have the ability to send email, they are not well configured to *receive* email. Therefore, we suggest that you do not send email directly from these machines. Instead, you should use a web browser to access your normal email account and send from there. The web browser should allow you to attach files directly from your account on the departmental computer.

In most assignments, you will need to turn in multiple files. At least one of these files should be your primary report. Computer programs should be in separate files with clear headers. If you end up needing to submit the homework more than once, please indicate that clearly in the subject line of the email.

Your homework must be emailed by 10 pm the night before class. Late homework will earn only 50% credit. If you are partially done, you should turn in what you have at 10 pm. If you turn in more of the assignment later, you will get 50% credit on the additional portion of your grade.

We **strongly** suggest that you not get into the habit of turning in late homework. The material in Physics 305 is highly cumulative. If you are falling behind, do not wait for long to seek help!

Your graded homework will be returned to you with feedback about your codes. Be sure to read the feedback, as it will contain information about what you have done wrong and suggestions on how to correct the problem in the future.

1.5 Homework 0

Due Wednesday, August 26, 10 pm

This is just a practice assignment. Its main purpose is to make sure that you have mastered logging in to the computers, downloading material, editing files, and turning in your homework. Once that is out of the way, we can begin serious work.

The assignment is to edit a short questionnaire, inserting answers to the questions, and then mail the result to the correct place. For section 1 (Toussaint), the correct place is “p305@physics.arizona.edu”. For section 2 (Eisenstein), the correct place is “miao@physics.arizona.edu”.

This assignment is due before 10:00 PM on Wednesday, August 26. Your email is automatically time stamped, and submissions sent after that will not be accepted.

Use a web browser to download (copy into a file in your directory) <http://www.physics.arizona.edu/~doug/questions.hw00>. To save this as a file, use the “Save as” option in the menu produced by the “File” button in firefox or konqueror. When the menu pops up, be sure to say that you want the result saved as a “text” file, rather than PostScript or source. Give the file some recognizable name. To do this, click on the filename, use “backspace” to delete the part of the name that you don’t want (usually the part following the last slash), and type in the desired name.

In the PAS 272 section, note that you should be running the web browser on faraday (type “firefox ...” in the SSH window, rather than the browsers on your windows machine. That way, when you download the file it will end up on faraday.

Now edit the file, using either “vi” or “emacs”.

Then mail it to “p305@physics.arizona.edu” or miao@physics.arizona.edu”. You can use a web browser to access your normal email account, and upload the file as an attachment.

Alternatively, in PAS 272, you can send mail from the Linux environment, for example with the “pine” command. This is simpler, but you should be warned that you need to be careful, since you might not get a warning if you send the email to an incorrect address. (*This is believed to be fixed on faraday 8/24*)

In addition to giving practice with editing and mailing, the purpose of these questions is to help us include the necessary material in the course while avoiding material that everyone already knows. In this respect, the question about the math courses you have had is the most important.

Please practice by composing and mailing the message to yourself and making sure that it contains what you mean for it to contain and nothing else. Then, when you have mastered the process, mail it to us.

1.6 A First C Program

Let's begin by considering the following program:

```
1  #include <stdio.h>
2  #include <math.h>
3  int main()
4  {
5      printf("Hello world!\n");
6      return(0);
7  }
```

Type this (omitting the line numbering) into a file `hello.c` using `emacs` and save it. We would like to have the computer execute this program. First, we must translate these words into commands that the computer can understand. This is done by “compiling” the program. You can do this by issuing the command “`gcc hello.c -o hello -lm`”. This will produce a file `hello` that is an “executable program”, meaning that the computer can now use this program like any other command². To execute the program, type `hello` at the command line. The computer will respond `Hello world!` Follow the instructions in the handout `first_c.txt`.

With the details of how to compile and run a program out of the way, let's now look at what the lines in the program mean.

Lines 1 and 2 are special, as indicated by the fact that they start with the pound sign (`#`). These are “preprocessor directives”. Before the compiling *really* starts, a pre-compiler goes through to interpret these lines. In particular, these lines enable us to use function calls from certain system libraries (namely `stdio.h` and `math.h`). A library is just a collection of functions; for later reference, the functions of the standard system libraries are described in Appendix B of Kernighan & Ritchie. Because you will so often use functions from these two libraries, you might as well include these two lines at the top of all of your programs³.

Next, line 3 begins the definition of a function. This function is called `main` and will return a variable of type `int`. The name `main` is special, because invoking a program from the command line is equivalent to calling the `main` function. All programs must have a function named `main` and you can think of this as the top level of the program. It is conventional for `main` to return a variable of type `int`; the compiler will issue a warning if it does not.

²I should explain that the `-o` option in the command line signals that the next word will be the name of the executable file. It is common for this to be given the same name as the source file, but without the `.c` suffix. Be careful not to make your output file the same as your source file, because you'll overwrite your source file!

³This is slightly bad advice, because programming convention is to only include libraries you actually use. But for now, let's do the simple thing! Also, the `-lm` option in the `cc` command line is needed if you want to include the `math.h` library.

Lines 4 and 7 contain a matched pair of curly braces. In C, braces are used to group a set of statements together (the resulting set is called a “block”). In this case, the statements between the braces are the body of the `main` function. In other words, calling `main` will execute these statements, starting from the top.

Lines 5 and 6 are the body of the `main` function. Line 5 calls a new function `printf`, which prints a line to the screen. Normally if you call another function, you must supply the definition of that function. In this case, `printf` is defined in the system library `stdio.h` that we accessed in line 1. Line 6 uses the C keyword `return`. This causes the function to end and return to the function that called it, which in this case is the command-line shell. The argument of the `return` statement must be the same type as the function. In this case, `main` was of type `int`, so we’ll return 0.

Note the semicolons (;) at the end of lines 5 and 6! All C statements must end with semicolons. However, the block delimiters (the curly braces) do not end with a semicolon; you can think of the `}` as carrying an implicit semicolon. Preprocessor directives (lines beginning with pound signs (#) such as lines 1 and 2) do not end with semicolons. Finally, lines beginning the definitions of functions (such as line 3) do not end with semicolons; you will learn more about this below!

Notice that the output indicated in the `printf` statement is contained in double quotes ("). Single quotes (') are used for a different purpose! The `\n` should be thought of as a single character and indicates that the computer should begin a new line of output. See what the output of the program looks like if you remove the `\n`!

Blank lines and extra spaces are ignored in C (unless inside double quotes), so you should feel free to add them as desired to make your program more readable. For example, a blank line between lines 2 and 3 separates these rather different aspects of the program.

Physics 305: Ch. 2

Getting started with C

2.1 An Introduction to C

In chapter 1 you wrote, compiled and ran your first C program. In this chapter we will spend some more time discussing the C Programming Language.

2.1.1 Variables and Declarations

Here's a more complicated program.

```
1  #include <stdio.h>
2  #include <math.h>
3  int main()
4  {
5      double b, product;
6      int j;
7      b = 4.5;
8      j = 3;
9      product = b*j;
10     /* We've formed the product; now output it */
11     printf("The value of b times j is %f!\n", product);
12     return(0);
13 }
```

Lines 1–4 and 12–13 are the same as before. Lines 5 and 6 define variables that we can use in the program. The variables `b` and `product` are defined to be of type `double`, which means that they are floating-point numbers (in other words, real non-integral numbers). The variable `j` is defined to be of type `int`, which means that it can only hold an integer. Note that the comma-separated list in line 5 can be used to define several variables of a given type on one line.

Lines 7 and 8 assign values to `b` and `j`, while line 9 assigns the product of `b` and `j` to the variable `product`. You see the format: the right-hand side of the equation is evaluated and assigned to the variable on the left-hand side. Again, all these statements must end with semicolons. Note that multiplication is accomplished by an asterisk (`*`); division is done with a slash (`/`).

Line 11 prints the value of the variable `product`. Try it. How does the compiler know how to distinguish this kind of `printf` from the one in the first program? The value `%f` in the output string is a signal to output a floating-point number. The compiler uses the next argument in the `printf` call to fill in the needed value.

Line 10 is called a “comment”. This is a line that is ignored by the compiler; it is only there as an explanation to humans reading the code. Comments begin with `/*` and end with `*/`. Don’t forget the closing `*/`! An alternative way to write comments is that `//` starts a comment and all the rest of the text on the line will be ignored; there is no closing mark for this syntax. It is very important that you include comments in code that you write so that you and any others (like your grader) who read your code can understand what the code is supposed to do!

Any variables you use in your programs **must** be defined before use. Usually this means that lines such as 5 and 6 must appear at the top of your functions, but we’ll learn more about this as we go along.

You can name your variables just about anything you like, but it is best to stick to letters, numbers, and the underscore (`_`) character. C does distinguish between upper and lower case, so `product` and `Product` are different variables. It is good practice to give names that explain what the variable is used for.

2.1.2 Loops and conditions

If all of our programs were executed sequentially in the order of the lines, we wouldn’t be able to do particularly complicated things. So here’s an example to start doing fancier control:

```

1  #include <stdio.h>
2  #include <math.h>
3  int main()
4  {
5      double input, cube;
6      int j;
7      input = 4.5;
8      cube = 1.0;
9      for (j=1; j<=3; j=j+1) {
10         cube = cube*input;
11     }
12     /* We’ve formed the cube; now output it */
13     printf("The cube of %f is %f!\n", input, cube);

```

```

14     /* Next, perform a test.  */
15     if (cube>100.0) {
16         printf("%f is greater than 100.\n", cube);
17     } else {
18         printf("%f is less than or equal to 100.\n", cube);
19         printf("Try making the input variable larger.\n");
20     }
21     return(0);
22 }

```

As you may have guessed from the names of the variables, the goal here is to calculate the cube of 4.5.

The first new concept in this program is the `for` command in line 9. This executes the statements in braces (here, just line 10) an unknown number of times, subject to the control of the criteria specified in the parentheses. Looking inside the parentheses, we see three statements separated by two semicolons. The first statement `j=1` sets the initial condition of the loop. The second statement indicates the test to be performed; when this test is failed, the loop will be ended and the program will continue with line 12. The third statement is the command to be performed after each time through the loop.

In other words, we begin by setting `j` to be 1. Since 1 is less than or equal to 3 (that's what `<=` means), we execute line 10. We then execute `j=j+1`, so `j` is now 2. Since $2 \leq 3$, we execute line 10 again and then do `j=j+1`. Again $3 \leq 3$, so we do line 10 and `j=j+1` once more. Finally, 4 is not less than or equal to 3, and so the loop ends and we proceed to line 12.

It is important to note that in line 10 (`cube = cube*input`), the right-hand side is evaluated completely before being assigned to the left-hand side. So we don't overwrite the value of `cube` before we use it!

Of course, for such a simple loop, we could have just as well replaced lines 8 to 11 with

```
cube = input*input*input;
```

However, the value of loops is in the ability to execute statements many times (far more than you'd care to type) and in the ability to not specify the number of iterations ahead of time. You will soon be writing programs in which the number of iterations depends on numeric values that the program develops only in the course of its execution.

The second new concept is the `if` command in line 15. This is fairly straightforward: if the comparison inside the parentheses is true, then the first block of statements (line 16) is executed; if not, then the second block (lines 18 and 19) is executed. After this, the program will continue on the line after the `if...then...else` statement, in this case line 21. Also, I should note that the `else` statement block

can be omitted if you simply want to execute a statement if something is true and do nothing if it is false. Here, we would replace lines 17 to 20 with a simple `}` (to close the open brace on line 15) if we wanted to print nothing if `cube` were less than 100.

In this case, we were testing an inequality, but this is a good time to tell you that testing an equality has a surprising syntax. Rather than a single equal sign, you use two! In other words, to test whether two variables `i` and `j` are equal, you use `if (i==j)`. A single equal sign is reserved for assignment not comparison, so using `if (i=j)` will set `i` equal to the value of `j` and then evaluate to false if `j` is equal to 0 and true otherwise, probably not what you want!

Note that the `printf` command in line 13 is a bit more complicated as well. Now the output string has two `%f` commands, which are filled in order from the second and third arguments to `printf`.

Finally, in both of these cases, you see the use of braces to group commands into blocks. It is not a problem that these braces appear inside of the ones delimiting the `main` function (lines 4 and 21). Each block is defined by the matching set of `{` and `}`; this is called “nesting”.

The pattern of indentation in the lines is only convention of C programmers. The program would work just as well if all of the leading spaces on each line were removed. However, it would be much harder to read, because you would not have the visual cues as to where the loops and conditional statements begin and end! It is important that you indent nested blocks in this manner. Four spaces is the typical amount; you will find that `emacs` tries to do this for you!

Technically, the braces can be omitted when a block of statements contains only one statement. In other words, we could write

```
for (j=1; j<=3; j=j+1)
    cube = cube*input;
```

or even more compactly

```
for (j=1; j<=3; j=j+1) cube = cube*input;
```

However, as soon as your block includes more than one statement (i.e. more than one semicolon!), you must use the braces.

2.1.3 Functions

A very important part of programming is dividing programs and problems into smaller pieces, known in C as functions. As in mathematics, functions have inputs and outputs. The function doesn't care about the origins of its inputs or the fate of its outputs. This means that you can use a function many times, each time with a different input or a different use for the output. It becomes a tool that other programs

or functions can use without knowledge of its inner workings. This is not only very economical but leads to code that is far easier to read and modify.

Let's see a simple example.

```
1  #include <stdio.h>
2  #include <math.h>
3  double cos_squared(double theta)
4  {
5      /* Given an angle in radians,
6         calculate the square of the cosine of the angle */
7      double cosx;
8      cosx = cos(theta);
9      return(cosx*cosx);
10 }
11 int main()
12 {
13     double input, output;
14     input = 4.5;
15     output = cos_squared(input);
16     printf("The square of the cosine of %f is %f.\n",input,output);
17     return(0);
18 }
```

Lines 3 to 10 define the new function. Line 3 specifies the name, the inputs, and the output. The name is `cos_squared` and it returns a value of type `double`, as specified by the first word on the line. The inputs are listed inside the parentheses. In this case, there is just one input, a floating-point variable that will be called `theta`. Note that line 3 does not end with a semicolon!

Lines 4 and 10 contain the braces that group the lines that define the function. Line 7 defines a new floating-point variable that we will use in our calculations. You do not need to define `theta` on this line; it has already been defined in the function declaration (line 3). However, it is critical to note that `theta` and `cosx` are the *only* variables you can use in this function. The variables `input` and `output` are visible only in the function `main` and not in `cos_squared`. Similarly, `theta` and `cosx` are not visible in `main`. This isolation is a feature, because it means that the action of one function cannot mess up another. All interaction between the functions is accomplished by the inputs and outputs of the function. We will have much more to say about this later.

Line 8 calls a function by the name of `cos`. As you might have guessed, this is a built-in function in the system library `math.h` that calculates the cosine of the given angle. This angle is interpreted to be in radians.

Line 9 does two things. The `return` statement specifies that the function will end, with control passing back to the place from which the function call was made.

However, because the `cos_squared` function is of type `double`, the second half of the `return` statement specifies the floating-point number with which to return. In this case, this expression itself requires calculation. As you can see, the calculation is simply the product of `cosx` with itself.

Notice that lines 5 and 6 is a comment that tells the reader what the function is going to do. It is very good practice to say this explicitly at the top of each function, although whether you do this before line 7 or line 4 or line 3 is up to you. Also note that the `/*` that began the comment and the `*/` that end it are on different lines.

Back in `main`, we see in line 14 how our new function is called. The expression in parentheses is evaluated and passed to `cos_squared` as the value of the variable `theta`. In this case, the expression is simple: just the value of the variable `input`.

Of course, in this program, the variable `output` doesn't have much use; we could have just have easily combined lines 15 and 16 into

```
printf("The sine of half of %f is %f.\n", input, cos_squared(input));
```

Take another look at the whole code and notice the parallels between the declaration of `cos_squared` and `main`. We have defined two functions, the first of type `double` and the second of type `int`. The first takes an input value of type `double`; the second takes no input. Also notice the parallels between the declaration of the functions and the declarations of variables, for example line 7. The format is the same: first the type, then the name.

Of course, as we said in the first section, the function `main` is special because that is where execution starts when the program is called from the Unix command line! This does beg the question: why didn't we reverse the order of the definition in the file so that `main` came first? The problem is that the name `cos_squared` is referred to in the body of `main`, and yet that name is not recognized by the compiler until the function `cos_squared` is declared. Note that the issue is *not* the order in which the names are encountered as the program executes, but merely the order in which the names appear in the source file!

There is a way to fix this. We need to alert the compiler to the name `cos_squared` before that name is used in `main`. This is done by specifying a prototype, which looks just like the statement that defines the function `cos_squared` in line 3 but which ends with a semicolon. Omitting the bodies of the functions themselves, we have:

```
1  #include <stdio.h>
2  #include <math.h>
3  double cos_squared(double theta);
4  int main()
5  {
6      ...
7  }
8  double cos_squared(double theta)
```

```

9  {
10     ...
11 }
```

Try it! Once line 3 has been given, the function `cos_squared` can be invoked anywhere subsequently in the file, even if it is not actually defined until the end of the file.

2.1.4 Input and Output

We'll have more to say about this later, but it is useful to know how to write to the screen and read from the keyboard. You've already seen the `printf()` function in action; here's another example:

```
printf("The value is %f and its square is %f\n", my_var, my_var*my_var);
```

Here the value of the second argument `my_var` is printed in the first `%f` slot, and the third argument is evaluated and printed in the second `%f` slot.

But let's say that the variable is a number like `my_var=5.0/3.0`. By default, the `%f` format will print 6 digits to the right of the decimal place, no matter how many digits are to the left. You might want something different, e.g., to show more or less precision. This can be done with a format like `%6.3f`, which will show exactly 3 digits to the right of the decimal point and at least 6 characters over all (counting the decimal point and the minus sign, if present).

Or perhaps the number is really big or really small? The format `%e` will print in scientific notation, so that the number 1234.56 will print as `1.234560e+03`. You can modify the number of digits here too; for example, `%8.3e` will print `1.235e+03` (note how it rounds the number).

If you have an integer variable, you can print this with the `%d` format. The format `%5d` will print at least this many spaces, so the number 123 will be printed as two spaces and then 123. This is useful for making columns of numbers line up nicely.

There are many other possible format statements, but this will get you going.

Next, let's consider how to read a number from the keyboard. This is done with the `scanf()` function. For example, the code

```
float mass;
printf("Please enter the mass of the ball in kg: ");
scanf("%f", &mass);
```

would print this line to the screen and then wait for you to type a number and hit return. The number would be placed in the `mass` variable. Note the ampersand `&` before the variable name; this must be present, but we'll learn why later.

Although `scanf` and `printf` look similar, they're not exactly the same. Whereas `printf` can have arguments that are expressions to be evaluated, `scanf` doesn't work that way. For example,

```
scanf("%f", &mass-1);
```

won't read your number and add one to it, then assign the result to the `mass` variable. We'll learn what this statement really means later on, but for now keep it simple. Also, `scanf` doesn't really handle user error in the inputs very well. For example, if you typed "the mass of the ball is 5 kg", `scanf` will not figure out what you mean. There are fancier ways to get and parse user input, but for now some simple `scanf` statements will allow you to control your programs by typing.

A common error with `scanf` is that the format *must* match the type of the variable. Use `%f` for float, `%lf` for double, and `%d` for int. We'll learn why later.

If you want to be sure about what the computer has read, you can always print out the value after the `scanf`:

```
float mass;
printf("Please enter the mass of the ball in kg: ");
scanf("%f", &mass);
printf("Ok. The mass of the ball is %f kg\n", mass);
```

That kind of caution might save you a lot of time!

2.1.5 Arrays

Often in numerical computing, one wants to store a set of related numbers. We don't want to give each of these numbers a separate variable name; there might be thousands, and at minimum we want the nomenclature to suggest the relation between the numbers. C provides this capability by "arrays", which you can think of as vectors.

Consider the following program that calculates and stores the first 10 triangle numbers (1, 3, 6, 10, 15, etc.).

```
1  #include <stdio.h>
2  #include <math.h>
3  int main()
4  {
5      double my_array[10];
6      int j;
7      /* First, load the array with the values 1..10 */
8      for (j=0; j<10; j=j+1) {
9          my_array[j] = (double) (j+1);
10     }
11     /* Now cumulate the array, so that each value is the sum of
12        the previous set */
13     for (j=1; j<10; j=j+1) {
14         my_array[j] = my_array[j]+my_array[j-1];
```

```

15     }
16     /* Finally, print out the array */
17     for (j=0; j<10; j=j+1) {
18         printf("%f ", my_array[j]);
19     }
20     printf("\n");
21     return(0);
22 }
```

Line 5 defines the array `my_array` with 10 elements. Note that square brackets are used in array syntax. There are now 10 floating-point variables, called `my_array[0]`, `my_array[1]`, ..., `my_array[9]`, available for use. You will of course have noticed that these indices run from 0 to 9 instead of 1 to 10. The reason for this convention will be clear later in the course¹. It is usually an error to refer to indices outside of the defined range, for example, `my_array[10]` or `my_array[-3]`. However, the compiler will not complain! Instead, the program will either crash when executed with a “segmentation fault” error or worse will generate garbage answers! So, when dealing with arrays and especially when combining arrays with `for` loops, be careful that you’re not referring to indices outside of the defined range.

Lines 8 through 10 use a `for` loop to access each element of the array and fill it with a value. In this case, we’re filling it with the values from 1 to 10. Note that the first section of the `for` command starts with 0 and that the second section tests $j < 10$ not $j \leq 10$ so that the loop will be exited before index 10 of the array is used.

Lines 12 to 14 use a `for` loop to further manipulate the array. The goal is to make the j th element equal to the sum of the numbers 1 to $j + 1$. You should convince yourself as to why the given code does this! Note that we began this loop with $j=1$ so as to avoid referring to `my_array[-1]`.

Finally, lines 15 to 18 print out the array on a single output line. Again, we use a `for` loop to print out each individual element. Note that we have not included the `\n` in each `printf` statement, because we do not want to start a new line each time. Instead, we include a trailing space for each one and then use one more `printf` after the loop to end the line.

As a side note, statements such as $j = j + 1$ appear so frequently in C that a special abbreviation has been created. `j++` by itself will increment the variable `j` by 1, whereas `j--` will decrement the variable by 1. So one often sees `for` loops such as

```

for (j=0; j<10; j++) {
    ...
}
```

You may have noticed that the number 10 appears four different times in the code in this section. This is an accident waiting to happen: you may decide at some point

¹The lingo is that C follows a “zero-offset” convention rather than a “unit-offset” one.

that you want to generate the first 15 triangle numbers and then forget to change one of the instances of 10. A way to limit this problem is to give the number 10 a new name by using a new preprocessor directive:

```
#define ARRAYSIZE 10
```

Put this line near the top of the file, say between lines 2 and 3, and then change every appearance of the number 10 to the word `ARRAYSIZE`. The program will work as before. What is really happening here is that, before compilation, the preprocessor goes through your file and changes every instance of the text `ARRAYSIZE` to the number 10 (just like a search-and-replace in a word processor). Hence, you get the same code as before, but now if you want to change the array size to 15, you need only change the `#define` line (and best yet, it's at the top of the file where you can find it). It is best to give `#define` labels very obvious names (like the use of all uppercase in this example), so that you don't get confused. Finally, note that there is no semicolon to end the `#define` line and that the new name `ARRAYSIZE` is *not* a variable and cannot be changed by commands in the code itself.

2.1.6 When things go wrong

Everyone makes mistakes, and most programmers spend most of their time trying to fix mistakes in their code. While we may all hope to make fewer mistakes, the more valuable skill is become faster in diagnosing and fixing the mistakes we do make.

Broadly speaking, there are three types of errors, in increasing order of horror and frustration.

Compile-time errors (syntax errors) These are errors in your source code that cause the compiler to be unable to translate your statements into instructions the computer can understand. Perhaps you omitted a semi-colon, used a variable name that hadn't been declared, mistook the type of a variable, or have a mismatched set of braces, parentheses, quotation marks, or comment delimiters.

When the compiler gets confused, it will print an error message that tries to explain why it can't understand a line. Importantly, this message will tell you the line number of the offending statement. Look at that line to try to see what's wrong. Check whether any of your punctuation is mismatched or missing. Check whether the variables you use are defined and are of the correct type.

Unfortunately, once the compiler has gotten confused, it will usually remain confused. When a line in your file is out of place, often many of subsequent lines also make no sense. Hence, instead of getting only one error message about the bad line, you'll also get dozens of errors about lines that are correct. *My advice is to start at the top of the error list and fix the first error, then try to*

recompile. You will often find that fixing the first error makes many of the rest go away.

The compiler may also issue “warnings”. These are not errors, in the sense that the compiler has successfully produced an executable file. However, the compiler is warning you that something looked odd albeit not outright illegal. This can be a sign of trouble ahead.

If you work through the errors systematically, top-to-bottom, you will usually be able to clear your code of compile-time errors in a reasonably short amount of time. If you are having problems, please talk to one of your instructors or classmates.

Run-time errors There may be errors in your code that the compiler can’t find but that will crash the program when you run it. For example, you may divide by zero, take the square root of a negative number, try to access memory that you haven’t reserved (e.g. by referring to an array out-of-bounds), or try to read from or write to a file that you haven’t opened.

These can be frustrating because it may not be clear where the program crashed. The first step is to figure out where. For now, you can do this by including extra `printf` statements, so that you can see how far the program is getting and what the values of relevant variables are at that point.

Warning: in Unix, there is a major gotcha lurking here! When you execute a `printf` command, your program generates some output. However, before this output is displayed on your screen, the operating system will “buffer” it, meaning that it saves up the output until there is a large enough amount and then writes to your screen all at once. This is done for efficiency. However, if the program crashes, anything in the buffer at that time is simply discarded. This means that your diagnostic `printf` statements may generate output that you never see, leading you to believe that your program crashed at some earlier point! One way around this is include `fflush(NULL);` statements after your `printf` diagnostics. This will force the operating system to empty (“flush”) any buffers before the program continues. I will teach you better ways to avoid this problem later in the term.

Another warning: on some systems, doing an illegal mathematical operation—for example, dividing by zero or taking the square root of a negative number—doesn’t crash the program but instead generates a NaN. NaN is short for “not a number”. The problem here is that the computer will keep happily propagating the NaNs: `5+NaN` equals NaN, `cos(NaN)` equals NaN, etc. Your program output will simply include NaN where you wanted answers. You are then faced with the problem of figuring out where the NaNs came from! See the next item.

Garbage time! Your program may not crash, but it may not do what you want

either. Of particular worry to the numerical physicist, it may not be computing the correct numbers!

The first challenge here is discovering the problem. You *must* study the outputs to see if values are what you expect. Print out intermediate steps in the calculation to see if those make sense. Read your code to try to assure yourself that it's doing what you intend.

This is very much akin to your experience in problem solving. When you compute an answer to a physics problem, you know that you must whether the result has the expected sign or is of the expected order of magnitude. Now you must be even more careful, because the computer is doing all of the intermediate steps.

Once you suspect that a result may be wrong, you have to investigate. Compute the intermediate steps. Carefully read the relevant code. Be suspicious. One useful tool that is easy to do with computers but hard with pencil-and-paper is that you can easily experiment with different sets of input values to try to diagnose the problem. Sometimes the error will be such that certain values will give correct answers while others won't, in such a way as to give you a clue. Better yet, there may be input values that simplify the problem so that you know what the answer should be. Just be sure that those special values didn't hide the error as well!

Learning to diagnose and fix these kinds of problems requires lots of practice. Be prepared for some frustration, but also try to learn from past mistakes!

Avoiding and fixing errors are the most compelling reason for writing readable, well-commented code and for learning to subdivide larger problems into a number of simpler subproblems that can be coded into individual functions. Most conventions in coding format, principles of code subdivision, and even aspects of the language itself are attempts to reduce the number of errors introduced into the code and to make them easier to find.

This doesn't mean that you need to put a comment for every line, but you should try to make sure that each separate logical step in an algorithm is somehow identified and documented. Using variable names that carry meaning is another helpful practice. Once you get used to C, a good code should read like a simple recipe.

2.1.7 Learning more C.

I hope that these examples will get you started. You should take the opportunity to experiment; see what works and what doesn't. At this point, you can't hurt the computer or your files, so just play around. If you have a code that works and want to try changes, you may want to copy the working copy to a new file so that you can always get back to some baseline working version.

You will of course need to learn more of the C language as you progress through this course. I have omitted some major topics from this introduction; we will introduce them later.

2.1.8 More C examples.

The following pages contain several other examples of simple C programs. Here, we look at two problems, each worked several different ways to give examples of different features and syntax. The first problem is to print out the first 10 factorials; the second is to look at the convergence of the Taylor series for $\sin(x)$.

```
1  /* Example 1a */
2  /* This simply prints out the first N factorials. */
3
4  #include <stdio.h>
5  #include <math.h>
6
7  int main() {
8      int N, n;    /* C is case-sensitive, so these variables are different */
9      int factorial;
10     N = 10; /* This is how many we will print */
11     factorial = 1; /* We start at 1 */
12     for (n=1; n<=N; n++) {
13         /* We're going to count up in n, holding the cumulative multiplication
14          * in the variable factorial */
15         factorial = factorial*n;
16         printf("%2d! = %7d\n", n, factorial);
17         /* We're printing out two variables here. We use %d for integers.
18          * The numbers indicate the number of characters used to print the values.
19          * \n signals for a carriage return.*/
20     }
21     return(0); /* This line is optional, but it is good form */
22 }
```

```
1  /* Example 1b */
2  /* This simply prints out the first N factorials. */
3  /* This version is just like Example 1a, but it uses a preprocessor
4     definition. */
5
6  #include <stdio.h>
7  #include <math.h>
8
9  #define MAX_N 10
10 /* This defines a simple search-and-replace equivalence. Now whenever
11    * we use the string 'MAX_N', it will substitute 10. This can be useful
12    * when there are many instances of the number and one doesn't want to
13    * forget to change some of them. It is NOT a variable, e.g. you cannot say
14    * MAX_N = 20;
15    * in the program. */
16
17 int main() {
18     int n;
19     int factorial;
20     factorial = 1; /* We start at 1 */
21     for (n=1; n<=MAX_N; n++) {
22         /* We're going to count up in n, holding the cumulative multiplication
23            * in the variable factorial */
24         factorial = factorial*n;
25         printf("%2d! = %7d\n", n, factorial);
26     }
27     return(0); /* This line is optional, but it is good form */
28 }
```

```
1  /* Example 1c */
2  /* This simply prints out the first N factorials. */
3  /* Here we'll use a function.  Actually, in this case, this is
4   * inefficient, because we don't get to re-use the previous multiplications,
5   * but it's good to illustrate the function. */
6
7  #include <stdio.h>
8  #include <math.h>
9
10 int factorial(int N) {
11     int n, result;
12     /* Note that these variables n and N are different from those in main() */
13     result = 1; /* We start at 1 */
14     for (n=1; n<=N; n++) {
15         /* We're going to count up in n, holding the cumulative multiplication
16          * in the variable result */
17         result = result*n;
18     }
19     return result;
20 }
21
22
23 int main() {
24     int N, n; /* C is case-sensitive, so these variables are different */
25     N = 10; /* This is how many we will print */
26
27     for (n=1; n<=N; n++) {
28         printf("%2d! = %7d\n", n, factorial(n));
29     }
30     /* Since we are only using a single statement in this block, we could
31      * have omitted this set of curly braces. */
32
33     return(0); /* This line is optional, but it is good form */
34 }
```

```
1  /* Example 1b */
2  /* This simply prints out the first N factorials. */
3  /* This version uses an array to store all of the values before we print them */
4
5  #include <stdio.h>
6  #include <math.h>
7
8  #define MAX_N 10
9
10 int main() {
11     int n;
12     int factorial[MAX_N+1];
13     /* Arrays of size N are indexed from 0 to N-1 */
14     /* We'll go ahead and define one extra spot, so that factorial[j]
15        stores j! */
16     factorial[0] = 1; /* 0! = 1 */
17     for (n=1; n<=MAX_N; n++) {
18         /* We're going to count up in n, holding the cumulative multiplication
19            * in the array factorial */
20         factorial[n] = factorial[n-1]*n;
21     }
22     for (n=0; n<=MAX_N; n++)
23         printf("%2d! = %7d\n", n, factorial[n]);
24     return(0);
25 }
```

```

1  /* Example 2a */
2  /* This computes the Taylor series approximation to sin(x) out to N terms
3   * and prints out the results, along with the error. */
4
5  /* Remember that sin(x) = x - x^3/3! + x^5/5! ...
6   * We will use the fact that the next term is the last one times
7     -x^2 divided by (2n-1)(2n-2), where n is the number of the term.
8   * Check that you see why that is. */
9
10 /* Of course, when one calls the function sin(x); the computer isn't just using
11  * the brute force Taylor series... */
12
13 #include <stdio.h>
14 #include <math.h>
15
16 int main() {
17     int N, n;    /* C is case-sensitive, so these variables are different */
18     double x;    /* This is the value we want to compute sin(x) of */
19     double full_answer, approx_answer, last_term, next_term;
20
21     x = 2.9;
22     N = 9;    /* This is how many terms we will use */
23     full_answer = sin(x);
24     printf("The correct answer of sin(%f) is %f\n", x, full_answer);
25
26     approx_answer = x;    /* The first Taylor series term */
27     last_term = x;    /* The next term will be derived from the last one */
28     for (n=2; n<=N; n++) {
29         /* Notice that we start with n=2; we've already done the first
30          * term before we enter the loop */
31         next_term = -last_term*x*x/(2*n-1)/(2*n-2);
32         /* Notice that we did not write /(2*n-1)*(2n-2).
33          * C doesn't parse the division sign the same way you're use
34          * to in algebra. */
35         approx_answer = approx_answer + next_term;
36         last_term = next_term;
37         /* We have to make sure the loop is ready for the next cycle! */
38         printf("Term %1d is %12.5e. The series is %10.7f, an error of %10.3e\n",
39             n, next_term, approx_answer, full_answer - approx_answer);
40         /* Several things here:
41          * 1) We use %f to print out results with fixed decimal place,
42          *    i.e. 12.3456, but %e to use scientific notation, i.e.
43          *    1.23456e1.
44          * 2) We can do math operations in the arguments of a function,
45          *    as we do here in the last argument.

```

```
46         * 3) We can have a line break between the first argument of
47         *   printf and the rest of the arguments. This was purely to
48         *   help the lines be readable. You can put line breaks anywhere
49         *   you want so long as it is not between the quotes or somewhere
50         *   where a space would change the meaning, e.g. in the
51         *   middle of a variable name, or between a function name and
52         *   its opening parenthesis. */
53     }
54     return(0); /* This line is optional, but it is good form */
55 }
```

```

1  /* Example 2b */
2  /* This computes the Taylor series approximation to sin(x) out to N terms
3   * and prints out the results, along with the error. */
4
5  /* Remember that sin(x) = x - x^3/3! + x^5/5! ...
6   * We will use the fact that the next term is the last one times
7     -x^2 divided by (2n-1)(2n-2), where n is the number of the term.
8   * Check that you see why that is. */
9
10 #include <stdio.h>
11 #include <math.h>
12
13 int main() {
14     int N, n; /* C is case-sensitive, so these variables are different */
15     double x; /* This is the value we want to compute sin(x) of */
16     double full_answer, approx_answer, this_term;
17
18     x = 2.9;
19     N = 9; /* This is how many terms we will use */
20     full_answer = sin(x);
21     printf("The correct answer of sin(%f) is %f\n", x, full_answer);
22
23     approx_answer = x; /* The first Taylor series term */
24     this_term = x; /* The next term will be derived from the last one */
25     for (n=2; n<=N; n++) {
26         this_term = -this_term*x*x/(2*n-1)/(2*n-2);
27         /* Unlike Example 2a, we've combined the last_term and next_term
28          * variables into a single variable. The right-hand side of the
29          * assignment is computed before the variable on the left-hand
30          * side is assigned. This means that we can use a variable and
31          * then overwrite its value in the same statement. In this case,
32          * we don't need the value of this_term except for this one
33          * computation, so we don't need a second variable to hold
34          * the result. */
35         approx_answer = approx_answer + this_term;
36         printf("Term %1d is %12.5e. The series is %10.7f, an error of %10.3e\n",
37             n, this_term, approx_answer, full_answer - approx_answer);
38     }
39     return(0); /* This line is optional, but it is good form */
40 }

```

```

1  /* Example 2c */
2  /* This computes the Taylor series approximation to sin(x) out to N terms
3   * and prints out the results, along with the error. */
4
5  /* This version is just like 2b, but we get the values for x and N from the user. */
6
7  #include <stdio.h>
8  #include <math.h>
9
10 int main() {
11     int N, n;    /* C is case-sensitive, so these variables are different */
12     double x;    /* This is the value we want to compute sin(x) of */
13     double full_answer, approx_answer, this_term;
14
15     printf("Please enter the value to be computed: ");
16     scanf("%lf", &x);
17     printf("Please enter the number of terms to be used: ");
18     scanf("%d", &N);
19     /* scanf() reads from the keyboard. This simple form will at least allow you to
20      * enter some values into your codes without recompiling. Unfortunately, it can
21      * behave badly; there are somewhat better but more complex ways to do this.
22      *
23      * Most important here is the use of the ampersand symbol (&) in the argument.
24      * This is because scanf requires the use of pointers, to be described later.
25      * Also, it is critical that the variable type of the argument match that listed
26      * in the string. Because x is a double, we have to use %lf (long float) rather
27      * than just %f. In printf, we could get away with either because we're not
28      * using pointers. Use %f for floats, %d for ints, and %c for characters. */
29
30     full_answer = sin(x);
31     printf("The correct answer of sin(%f) is %f\n", x, full_answer);
32
33     approx_answer = x; /* The first Taylor series term */
34     this_term = x;    /* The next term will be derived from the last one */
35     for (n=2; n<=N; n++) {
36         this_term = -this_term*x*x/(2*n-1)/(2*n-2);
37         /* Unlike Example 2a, we've combined the last_term and next_term
38          * variables into a single variable. Sometimes one can get away
39          * with this. */
40         approx_answer = approx_answer + this_term;
41         printf("Term %1d is %12.5e. The series is %10.7f, an error of %10.3e\n",
42              n, this_term, approx_answer, full_answer - approx_answer);
43     }
44     return(0);    /* This line is optional, but it is good form */
45 }

```

```

1  /* Example 2d */
2  /* This computes the Taylor series approximation to sin(x) out to N terms
3   * and prints out the results, along with the error. */
4
5  /* In this version, we get x from the user but compute terms until we get
6   * convergence. We define convergence as terms smaller than 1e-15, because
7   * we know that sin(x) varies between -1 and 1, so 1e-15 is very small.
8   * Defining convergence in other problems can be difficult; you may not know
9   * in advance how big the answer is or whether the truncated terms will in
10  * fact be adding up to a negligible value. */
11
12  #include <stdio.h>
13  #include <math.h>
14
15  int main() {
16      int N, n;    /* C is case-sensitive, so these variables are different */
17      double x;   /* This is the value we want to compute sin(x) of */
18      double full_answer, approx_answer, this_term;
19
20      printf("Please enter the value to be computed: ");
21      scanf("%lf", &x);
22
23      N = 100;
24
25      full_answer = sin(x);
26      printf("The correct answer of sin(%f) is %f\n", x, full_answer);
27
28      approx_answer = x; /* The first Taylor series term */
29      this_term = x;    /* The next term will be derived from the last one */
30      for (n=2; n<=N; n++) {
31          this_term = -this_term*x*x/(2*n-1)/(2*n-2);
32          /* Unlike Example 2a, we've combined the last_term and next_term
33           * variables into a single variable. Sometimes one can get away
34           * with this. */
35          approx_answer = approx_answer + this_term;
36          printf("Term %1d is %12.5e. The series is %10.7f, an error of %10.3e\n",
37                n, this_term, approx_answer, full_answer - approx_answer);
38          if (fabs(this_term)<1e-15) break;
39          /* The fabs() function takes an absolute value of a floating point
40           * number. Note that abs() takes the absolute value of an integer;
41           * this is usually not what you want and can be an annoying bug
42           * to find! */
43
44          /* The break command causes one to exit from the innermost loop,
45           * in this case the for loop. This means that we don't have to

```

```
46         * compute all N terms!  The program will continue executing after
47         * the loop. */
48     }
49     /* There are two ways we could have exited the loop.  We might have reached
50     * our convergence criteria and encountered the break statement.  Or we might
51     * have computed N terms and failed the n<=N test in the for statement.
52     * How can we tell?  Do a test of n vs N. */
53     if (n<=N)
54         printf("We converged after %d terms.\n", n);
55         /* Note that the variable n is still defined, even though we have left
56         * the loop */
57     else
58         printf("We computed %d terms, but did not converge.\n", N);
59
60     /* If our goal was to reach convergence, why have any limit on the number of
61     * terms at all?  It can be useful to guard against very long loops (or the
62     * so-called 'infinite loop').  Here we are using the for loop to count the
63     * number of iterations so that we can stop and warn the user when things are
64     * not converging well enough.  Poor convergence often means something is
65     * not behaving as expected.  For example, see what happens when you use x=50
66     * in this code.  The code will use about 83 terms and claim convergence, but
67     * answer is horribly wrong.
68
69     * What happened?  Some of the terms are getting very large and oscillatory.
70     * They should cancel, but the computer only has about 15 decimal places of
71     * accuracy.  When one adds 1020 to -(1020+1), one doesn't get the right
72     * answer.
73
74     * Hence, one might (if one were going to compute sin(x) in this way) choose
75     * to set a much smaller maximum number of terms, so as to guard against this
76     * kind of problem. */
77
78     return(0);    /* This line is optional, but it is good form */
79 }
```

```

1  /* Example 2e */
2  /* This computes the Taylor series approximation to sin(x) out to N terms
3   * and prints out the results, along with the error. */
4
5  /* This version is much like Example 2d, but we'll use a while loop */
6
7  #include <stdio.h>
8  #include <math.h>
9
10 int main() {
11     int N, n;    /* C is case-sensitive, so these variables are different */
12     double x;    /* This is the value we want to compute sin(x) of */
13     double full_answer, approx_answer, this_term;
14
15     printf("Please enter the value to be computed: ");
16     scanf("%lf", &x);
17     N = 100;
18
19     full_answer = sin(x);
20     printf("The correct answer of sin(%f) is %f\n", x, full_answer);
21
22     approx_answer = x; /* The first Taylor series term */
23     this_term = x;    /* The next term will be derived from the last one */
24     n=2;
25
26     while (fabs(this_term)>1e-15) {
27         /* In the while loop, the loop will execute so long as the
28          * test is true. */
29         this_term = -this_term*x*x/(2*n-1)/(2*n-2);
30         /* Unlike Example 2a, we've combined the last_term and next_term
31          * variables into a single variable. Sometimes one can get away
32          * with this. */
33         approx_answer = approx_answer + this_term;
34         printf("Term %1d is %12.5e. The series is %10.7f, an error of %10.3e\n",
35                n, this_term, approx_answer, full_answer - approx_answer);
36         if (n>N) break;
37         /* The break command works for while loops too.
38          * The program will continue executing after the loop. */
39         n++; /* We have to increment n explicitly. */
40     }
41     /* There are two ways we could have exited the loop. We might have reached
42     * our convergence criteria and encountered the break statement. Or we might
43     * have computed N terms and failed the n<=N test in the for statement.
44     * How can we tell? Do a test of n vs N. */
45     if (n<=N)

```

```
46     printf("We converged after %d terms.\n", n-1);
47     /* Tricky here: The last statement of the loop incremented n and THEN we
48     * did the test.  So n is one bigger than it was when we did the
49     * calculation of this_term.  These kinds of ending conditions always
50     * have to be thought through. */
51     else
52         printf("We computed %d terms, but did not converge.\n", N);
53
54     return(0);    /* This line is optional, but it is good form */
55 }
```

```

1  /* Example 2f */
2  /* This computes the Taylor series approximation to sin(x) out to N terms
3   * and prints out the results, along with the error. */
4
5  /* This version is much like Example 2d and 2e, but we'll use a do..while loop */
6
7  #include <stdio.h>
8  #include <math.h>
9
10 int main() {
11     int N, n;    /* C is case-sensitive, so these variables are different */
12     double x;    /* This is the value we want to compute sin(x) of */
13     double full_answer, approx_answer, this_term;
14
15     printf("Please enter the value to be computed: ");
16     scanf("%lf", &x);
17     N = 100;
18
19     full_answer = sin(x);
20     printf("The correct answer of sin(%f) is %f\n", x, full_answer);
21
22     approx_answer = x; /* The first Taylor series term */
23     this_term = x;    /* The next term will be derived from the last one */
24     n=1;
25
26     do {
27         /* In the do..while loop, the loop will execute so long as the
28          * test is true. However, the test is at the end of the loop,
29          * so the loop ALWAYS executes at least once. */
30         n++; /* We have to increment n explicitly. */
31         this_term = -this_term*x*x/(2*n-1)/(2*n-2);
32         /* Unlike Example 2a, we've combined the last_term and next_term
33          * variables into a single variable. Sometimes one can get away
34          * with this. */
35         approx_answer = approx_answer + this_term;
36         printf("Term %1d is %12.5e. The series is %10.7f, an error of %10.3e\n",
37              n, this_term, approx_answer, full_answer - approx_answer);
38         if (n>=N) break;
39         /* The break command works for do..while loops too.
40          * The program will continue executing after the loop. */
41         /* We use n>=N rather than n>N here (compare to Example 2e), because
42          * otherwise we would compute the 101'th term before realizing.
43          * Again, these kinds of loop orderings have to be thought through. */
44     } while (fabs(this_term)>1e-15);
45

```

```
46     /* There are two ways we could have exited the loop. We might have reached
47     * our convergence criteria and encountered the break statement. Or we might
48     * have computed N terms and failed the n<=N test in the for statement.
49     * How can we tell? Do a test of n vs N. */
50     if (n<=N)
51         printf("We converged after %d terms.\n", n);
52         /* Unlike Example 2e, we've exited with n being set correctly. */
53     else
54         printf("We computed %d terms, but did not converge.\n", N);
55
56     /* Why use a do..while instead of a while loop? It depends on whether you
57     * want to require the loop to execute at least once. In example 2e, if one
58     * entered x=0, then the loop would have been skipped entirely. */
59
60     return(0);    /* This line is optional, but it is good form */
61 }
```

2.2 Unix Concepts: Redirection and Piping

The Unix operating system gives every command that you execute one input stream and two output streams. By default, the input stream is simply taken from your keyboard, and both output streams are printed to your terminal window. The input stream goes by the name `stdin`. One output stream (`stdout`) is intended for normal output, while the other one (`stderr`) is intended for error messages. As examples, when you execute `ls`, your files are printed to `stdout`, which appears in your window; when you execute `rm`, the confirming question is printed to `stderr` and appears on your screen, while the answer you type is delivered to `rm` via `stdin`.

Unix gives you the powerful ability to reassign the behavior of these three input/output streams. You can “redirect” the output streams so that they are written to files rather than your screen, or “redirect” a file to provide the input stream rather than your keyboard. Better yet, you can use “pipes” to connect the output stream of one program to the input stream of another.

While these features have many complex applications, we want to focus here on the simplest ones.

2.2.1 Redirecting `stdout`

You have already written programs that print output to the screen. Often you would like to save this output in a file; for example, you might like to run different parameter sets or keep the output to give to a plotting program or include in a written report.

The command `myprogram > myfile` will execute the program `myprogram` and send standard output `stdout` to the file `myfile`. The right angle bracket (`>`) signals the redirection. Try an example:

```
% ls > filelisting
% more filelisting
```

The file `filelisting` now contains the names of your files.

If the output file already exists, then the file would normally be overwritten. However, I have configured your shell options so that existing files will not be overwritten; instead, the shell will inform you that the file already exists. To force a file to be overwritten, use `myprogram >! myfile`.

You also have the ability to append the output of a program to an existing file. In other words, the new output is simply added to the end of the target file, without disturbing its initial contents. Appending is done by `myprogram >> myfile`.

It is worth noting that these commands redirect `stdout` only. `stderr` is still printed to your screen. This can be useful: all your normal output can be going to your file, while any error messages are printed to the screen instead of getting lost in a file of output. It is also possible to redirect `stderr` to a file, but this is a more advanced topic that we won't need in this course.

2.2.2 Piping

Unix has many commands to do simple text manipulation, and it can be very useful to be able to link together chains of these commands. This is done by pipes. The pipe character is the vertical line (`|`).

For example, your program may produce lots of output (e.g. you may have put in extra print statements for debugging) and you would like to page through it with `less`. Then the command

```
% myprogram | less
```

will send the output of `myprogram` to the input to `less`, which then displays the output page-by-page as normal.

You can string more than one pipe together, e.g.

```
% myprogram | sort | tail
```

As you might guess, the output of `myprogram` is sent to the input of `sort`, the output of which is sent to `tail`.

You may recall that `less` normally acts on a file, e.g. `less myfile`. However, it is a common feature of Unix commands that if the file is omitted, `stdin` is used instead.

Some common Unix commands appearing in pipes include:

less As described in “An Introduction to Unix”, the `less` command prints the contents of a file a page at a time. Type the space bar to see the next page, type `b` to go back one page, type `q` to quit the program. `d` and `u` will go up and down half-pages, while `j` and `k` will go up and down line-by-line. `less` is an improved version of its famous cousin `more`.

head -20 This prints the first N lines of a program, where N is the number after then minus sign (here 20). If the `-N` option is omitted, the first 10 lines are printed.

tail -20 Just like `head`, but it prints the last N lines of a file.

sort Sorts the lines of a file. There are a complex set of options that control how the sorting proceeds (e.g. which columns), but one that you should know is that `sort -n` sorts in numerical rather than dictionary order.

wc Counts the number of lines, words, and characters in a file. The mnemonic is “word count”.

grep <pattern> This filters a file so that only lines containing the string `pattern` are outputted. There is a very complicated syntax (known as “regular expressions”) to govern the pattern matching, but as expected letters and numbers

match to themselves. In other words, `grep cosine myfile` will go through the file `myfile` and print any lines containing the string “cosine”. Beware that the period (`.`) matches any character, so searching for decimal numbers can give surprising results; use `\.` to match a normal period.

`tee <file_name>` This splits an output stream, sending one copy to the named file and the other copy to `stdout`, which can go to your screen or on to another program via a pipe.

2.2.3 Redirecting standard input

We will not need this at present, but for completeness, you can send a file’s contents to a program by redirecting with the left angle bracket (`<`). In other words, `myprogram < myfile` will execute `myprogram` and provide it with the contents of `myfile` as if you were typing it at the keyboard.

2.3 Homework 1

Due Monday, August 31, 10 pm

Assignment number 1 is to write, run and report on several simple programs. Remember that your homework submission is a report which should contain your programs, but must also explain what each program does, how you tested it, some sample input and output, and any conclusions that you reach. (In the future, you will also need to discuss the numerical methods that you use, but these will be trivial in this assignment.) Remember that your homework report should be the body of your email message or a plain text attachment (but not both). The graphs for questions 3 and 4 should be postscript files sent as attachments.

1. Write and test a program to convert a temperature in degrees Fahrenheit into a temperature in Centigrade. This program only needs to read one number, and print one number for its output.
2. Write and test a program to print the first twenty powers of two. That is, print the value of 2^1 through 2^{20} .
3. Write and test a program to print x and $\sin(x)$ for values of x from 0 to 2π in steps of 0.1. The output should be a bunch of lines of text, each containing a value of x and the corresponding $\sin(x)$. Use “`Philspplot`”, “`graph`”, or another program of your choice to make a graph of the output.
4. Demonstrate the convergence of a Taylor expansion. Find the Taylor expansion around $x = 0$ of $\exp(-x)$ up to fourth order in x . Write a program to make a table of the values of $\exp(-x)$ in the interval $0 \leq x < 1$. Then make data files (separate) of the Taylor series to first order, to second order, third order and fourth order. You may do this by editing the program for each order that you want or by using `scanf()`. Then plot the function $\exp(-x)$ and each of the Taylor expansions on the same plot.

