EECS 211: Simple Introduction to Using UNIX for Programming Assignments

Ioan Raicu
Spring 2010
iraicu@eecs.northwestern.edu

Adapted from L. Henschen, September 2007

MARCH, 2010

Table of Contents

Introduction

The purpose of this document is to provide students in this class with enough of an introduction to certain UNIX tools so that they can do their programming assignments. It is not meant to be a comprehensive or general introduction to UNIX features, nor is it intended to introduce operating systems concepts. The coverage of features for any individual UNIX tool is not deep but should be enough for students to use the tools effectively for this class. Students are referred to any of a large number of standard books on UNIX tools as well as the "man" command within UNIX itself for additional information on these and other tools.

The material presented here has been compiled from a variety of standard books on UNIX as well as from previous teachers and Teaching Assistants for this course, particularly Prof. Peter Scheuermann, Vana Doufexi, Olivier Ghica, and Hui Ding.

Many sections of this document illustrate actual dialogues with the UNIX system or the student's own PC or describe text that the student must enter into a field. To distinguish such information from the normal text in this document, the following color-coding scheme is used:

- Orange text is text the user types.
- Green text is text that the computer displays back to the user.

Software Required for Remote Login

Each student will be given an account for one of the department's UNIX labs, typically the T-lab. This account allows both on-site and remote login and provides non-volatile storage on the lab's file system. User files can be accessed from any machine in the lab or through remote login.

For security reasons, remote login to the UNIX machines in the department lab must be made using a secure shell client (SSH) protocol. Requests for connection through a non-secure software tool, such as HyperTerminal, will not be accepted. If you are accessing the lab remotely, you will need to go through Northwestern's virtual private network, for an added level of security. See

http://www.it.northwestern.edu/offcampus/index.html

for instructions how to add and then configure a connection to this private network

An excellent, free software package allowing secure remote login can be obtained from

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

Select the release version of putty.exe appropriate for your computer. (Note, this site also posts a so-called "development snapshot", which is their latest working version and which is not guaranteed to work.)

Use of the graphics debugger (GDD) remotely requires a computer which supports an X-Windows environment. If your computer does not support this, you can check the department lab web site for instructions on how to obtain the required software:

http://www.ece.northwestern.edu/CFS/PCs.html

Logging in and the man Command

Remote login is begun by establishing a connection to the Northwestern University Virtual Private Network and then executing the secure client network connection on your laptop or PC. The SSH software should pop up a window, and you should request it to make a remote connection. The SSH software will ask for several pieces of information at various points in the opening dialogue. Some of these will be in the initial panel when you open the software, others may be requested in later windows before the login is complete.

1. host name or IP address – type tlab-yx.cs.northwestern.edu, where xy is any of 11, 12, …, 18.
2. port – typically this should be set to 22 if it doesn't automatically default to that value
3. protocol – if your secure-client software offers a selection of protocols, be sure that SSH is selected
4. user name – this is your UNIX lab user account name
5. password – this is your UNIX lab user password

The putty SSH software mentioned in the previous section immediately opens a window in which the first three items are to be entered. When they have been entered or selected, type connect. The window will change, at which point you are connected to the department's UNIX lab. You typically need to press <ENTR> once or twice, after which the UNIX system will prompt you for your user name and password.

UNIX is a command-line system. Instead of clicking icons with a mouse the user types the name of the program to be executed and possibly also options and inputs for the program.

The UNIX system then displays a prompt, typically the name of the machine that you are running on plus the path to the current working directory and the percent sign. At this point you may type commands into the UNIX system. Two commands that you will use a lot are ls and man. The ls command will be described in the next section. The man command allows you to ask for help on how to use the UNIX commands. For example, you could type

```
cayman :~ % man man
```

to see (a rather long) description of the man command, how to use it and the various options you can use when using it. The description is displayed one screen at a time. If the description is longer than one screen you can get the next line of the description by typing <ENTR> or the next screen by pressing the space bar. For novice UNIX users, probably most of the description is not understandable and can be ignored.

When connecting to the department's UNIX lab you will be connected to one of the UNIX machines, but not necessarily the same one each time. You will still be able to

access any files you have on the system because the files are stored under your user name on the system file server and are non volatile.

Files and Directories

Files are organized into hierarchical structures, similar to windows folders. In UNIX terminology, a folder is called a directory. When you first login, you are "in your home directory". Your home directory is a directory whose name is your user name and which occurs, along with probably most other users' home directories, at a particular place in the UNIX file system hierarchy. You can see the full path of the current working directory by using the pwd (print working directory) command. For example:

```
cayman :~ % pwd
/a/core/files9/home/henschen
cayman :~ %
```

You can list the files in the working directory with the ls (list) command. For example:

```
cayman :~ % ls
  … list of files and directories …
cayman :~ %
```

It is important to understand that your directories will be listed along with your files and will be indistinguishable, unlike Windows which lists the folders as icons. You can use the "-l" option to list both a key character ('d' for directory, '-' for file) plus all the permissions associated with this file. (File permissions is an advanced topic beyond the scope of this document.) For example:

```
cayman :~ % ls -l
-rwx------ 1 henschen eecs    560 Jan 18 2007 a.out
drw-rw-rw- 2 henschen eecs  27560 Jan  6 2007 project1
-rw-rw-rw- 1 henschen eecs   5948 Dec 21 2006 xmaslist
cayman :~ %
```

The directory/folder character is the first one. The next nine characters list the permissions (r for read, w for write, x for execution). Following the number is the owner of the file (henschen), the group to which the owner belongs (eecs), the file size, the date the file was created, and finally the file or directory name itself. Unless directed otherwise, the ls command will list the items in alphabetic order. Many folders contain system files and other special files. These files have names that begin with '.'. You can have these displayed by the ls command by including the –a attribute on the command line.

It is wise to organize your files into related groups with directories (like folders in Windows systems). To make a new directory in the working (i.e., current) directory, use the command mkdir (make directory). In the opposite direction, you can delete a directory with the command rmdir (remove directory). For example:

```
cayman :~ % mkdir xyz
cayman :~ % ls -l
-rwx------ 1 henschen eecs    560 Jan 18 2007 a.out
drw-rw-rw- 2 henschen eecs  27560 Jan  6 2007 project1
```

```
-rw-rw-rw- 1 henschen eecs   5948 Dec 21 2006 xmaslist
drw-rw-rw- 1 henschen eecs    256 Jan 30 2007 xyz
cayman :~ % rmdir xyz
cayman :~ % ls -l
-rwx------ 1 henschen eecs    560 Jan 18 2007 a.out
drw-rw-rw- 2 henschen eecs  27560 Jan  6 2007 project1
-rw-rw-rw- 1 henschen eecs   5948 Dec 21 2006 xmaslist
```

The directory must be empty or UNIX will not remove it but print an error message. Individual files can be removed with the command rm (remove). For example:

```
cayman :~ % rm xmaslist
```

Navigation is accomplished with the cd (change directory) command. The most commonly used forms are:

- cd project1 project1 should be in the current working directory and becomes the new working directory
- cd .. the two dots mean go up one directory level
- cd go to the HOME directory

The command to make a copy of a file is cp. This command takes two arguments – the name of the file to be copied and the name of the new file. For example:

```
cayman :~ % cp main.cpp main0125.bak
```

The source file is a file in the current working directory, and the new file is created in the current working directory. You can specify a file in another directory by giving a path. A simple path is relative to the current working directory; a full path is relative to the root of the file system. The character '/' is used to separate multiple directories in a path. For example, the command

```
cp xyz/main.cpp mybackups/eecs211/project2/main0129.cpp
```

says to copy the file main.cpp from the xyz directory within the current working directory to the directory project2 within the directory eecs211 within the directory mybackups within the current working directory and give it the name main0129.cpp. You can move a file using the mv command, which has the same format as cp.

You can display all or selected parts of a normal file by using one of the following commands:

- cat *filename* display the whole file
- less *filename* display the whole file one screen at a time
- head *filename* display the first 10 lines of the file
- tail *filename* display the last 15 lines file

For files that are more than 20 or so lines long, cat is not a good choice because the text flies by too fast to read. Instead, use less and use the space bar to move to the next screen.

Finally, if you are not sure which of a number of files has some text you are looking for, you can use the grep command to look for a word or phrase in a file. The

grep command takes two arguments – a text to search for and a file to search in. For example:

```
cayman :~ % grep MAX_CHARS main.cpp
      char cmd_line[MAX_CHARS];
      for(j=0; j<MAX_CHARS; j++) {
cayman :~ %
```

The grep command is case-sensitive unless you specify the –i (ignore case) option. For example:

```
cayman :~ % grep max_chars main.cpp
cayman :~ % grep –i max_chars main.cpp
      char cmd_line[MAX_CHARS];
      for(j=0; j<MAX_CHARS; j++) {
cayman :~ %
```

You can also have the grep command list the line numbers of the lines that it finds by using the –n option.

```
cayman :~ % grep –n MAX_CHARS main.cpp
      char cmd_line[MAX_CHARS];
      for(j=0; j<MAX_CHARS; j++) {
cayman :~ %
```

Finally, you can search in several files by listing them on the command line.

```
cayman :~ % grep –n MAX_CHARS main.cpp system_utilities.cpp
```

To summarize, then, the most commonly used file commands are:

- pwd - print the path to the working directory
- ls - list the files in the working directory
  - –l option specifies long format for the output
  - –a option includes the '.' files
- mkdir - make a new directory
- rmdir - remove a directory (must be empty)
- cd - change to a new working directory
  - directory name
  - .. to move to the parent of the current working directory
  - no argument to move to your HOME directory
- cp - copy a file
- mv - move a file
- cat, less, head, tail – display contents of a file
- grep - search for a text string in a file
  - -i to ignore case
  - –n to print the line numbers

Compiling and Running C++ Programs - Introduction

This section provides a quick introduction to compiling and running C++ programs with the GCC (GNU Compiler Collection) on UNIX. A more complete discussion of how programs spread across multiple files are processed by GCC is given in a later section.

Simple C++ programs in a single file are compiled with the g++ command. For example:

```
cayman :~ % g++ hello.cpp
```

(Note, C programs would be compiled with the gcc command.) This command will cause the GCC compiler to analyze the program in the file hello.cpp. If there are no syntactic errors, the compiler will translate the program into an intermediate form. If your program used any functions defined in other files (e.g., C++ library functions like cin and cout, DLLs that you, etc.) and the intermediate form can be linked successfully with those, an executable file will be generated and given the name a.out. You may then execute your function by calling it from the command line using "./" in front of the file name:

```
cayman :~ % ./a.out
```

Any cout statements in your program will generate text in your login window; any cin statements will accept text from the keyboard. As with most C++ systems, text from the keyboard is not passed into the program until the user types <ENTR>.

If errors are encountered during any step of the process, appropriate error messages are displayed, and no executable file is created. If the errors were found during the compile phase (translation from C++ source to intermediate form), line numbers will be displayed along with the error message itself. If the error occurred during the linking phase, the linker will identify the source file name, if possible, which contains a problem.

As with most UNIX commands the user can specify options to change the default behavior of the compiler. The four most commonly used options with the command are:

- -Wall - (Warnings – all) display warning messages in addition to error messages;
- -o - (Output) name the executable file something besides a.out
- -ggdb - include the information necessary to use the Gnu debugger gdb
- -c - compile the source files but do not generate the executable file

Just because a program compiles and links does not mean it is correct. Many times the compiler will translate C++ code which is not strictly correct syntactically. Using the –Wall option causes the compiler to display a warning message for each such anomaly in the program. Programmers should read warning messages to make sure the warning isn't in fact an error or mistake in the code. C++ applications are normally developed with many cpp files, and often a programmer may focus on one or two of these at a time. Each cpp file will generate an output file when successfully compiled. If each execution of g++ stored the output in a.out, each successive compile would overwrite the output from the previous compile unless the user made a copy first. For safety and clarity, it is common to give the intermediate, compiled version of a file the same base name as the

file itself. This is done by including "-o fname" in the command line. Debugging will be discussed in a later section. Finally, it is often desirable to just compile a cpp file to see if there are any typos or syntactical errors without actually linking; this is especially the case when first typing in a new file.

These options can be used independently or together in the same command. The "-Wall", "-ggdb" and "-c" options come after the command and before the cpp file name, in any order; the "-o fname" comes at the end of the line. For example:

```
cayman :~ % g++ -Wall –ggdb hello.cpp –o hello
cayman :~ % ./hello
```

You may compile several cpp files together by listing them in a single g++ command line.

```
cayman :~ % g++ -Wall helloa.cpp hellob.cpp –o hello
cayman :~ % ./hello
```

In this case the executable produced would be named "hello".

REMEMBER: Header files are included by the C/C++ preprocessor (see later section). They are NOT mentioned on the gcc or g++ command line itself.

EMACS: A Simple Line Editor for UNIX and C++

EMACS is a simple line editor that can be used to enter and edit C++ files and header files. EMACS allows a user to create new files, edit a file, move around in the file using the cursor keys, search for text, and many other basic operations. It automatically indents subordinate blocks in c/cpp and h files. It is not a sophisticated, visual-based editor; however, it is sufficient for most purposes for this class.

You normally invoke emacs with the command name and a file name:

```
cayman :~ % emacs test.cpp
```

If the file already exists, it will be loaded. If it does not exist, it will be created. The emacs window normally consists of two parts – a buffer showing the contents of your file and a command line at the bottom of the screen. You type text into the upper area and use the arrows, Page Up/Down, and other common keys to edit and move around in the file. You type CTRL key combinations and ESC key sequences to request emacs to perform operations like save a file or exit. An example will be shown in class, and a pdf file containing a summary of all the emacs commands (including many more than you will use in this class) will be posted on Blackboard.

Here is a summary of the most useful CTRL and ESC commands. The notation C-*letter* means to hold the CTRL key while you type *letter*. For example, C-x means CTRL-x. On the other hand, ESC-*letter* means to press the two keys in sequence; press ESC, then release ESC, then press *letter*. C-*letter* followed by another key press means press the two combinations in order. For example, C-x C-c means press CTRL-x and then press CTRL-c.

Quit emacs:

- C-x C-c

Basic file manipulation:

- C-x C-s save
- C-x C-c quit
- C-x i insert another file into this file at the cursor position

Navigation in text

- Use the four arrows (up/down/left/right)
- ESC-< to go to the beginning of the file, ESC-> to go to the end
- C-v or scroll down one screen, ESC-v to scroll up one screen. PageUp and PageDown also scroll up and down.

Undo

- C-x u undo the last command or keystroke
- DEL delete text backwards
- C-d delete text forwards.

Search

- C-s begin dynamic search forward. You type text in the command line area at the bottom of the screen. Each additional letter typed moves the cursor to the next place in the file that matches the sequence of letters typed so far. In addition, emacs underlines all other occurrences of the search text in the file.
- C-r begin dynamic search backwards. Same except the search goes backwards.
- <ENTR> terminate the search
- Typing C-s or C-r before a search is terminated moves the cursor to the next (previous) occurrence of the search string.
- ESC-p/ESC-n emacs remembers the search strings the user has entered in a circular list. If at any time during the search you type one of these commands, emacs will display the previous or next search string. You can continue to change the search string, and then when you found the one you want press C-s or C-r to continue searching, now with the newly chosen search string.

Copy and Paste

- NOTE: You can use your mouse to identify a region of text, copy it into a Windows buffer, and then paste into any application expecting input from the keyboard. For example, you could open a file on your PC, use the mouse to copy a block of text, bring the window with emacs to the front, and then paste the text into your emacs buffer.
- Registers. emacs has a set of numbered registers, each of which can hold text.
    - You identify a block of text in two steps
        - Put the cursor at the beginning of the text and enter the command C-*space* (called "marking" in emacs)
        - Move the cursor to the end of the text and enter the command C-x r s *register_number* to save the text in the indicated register
    - You insert the text from a register with the command C-x r i *register_number*

Shell

- ESC-! execute a shell command. The user may type a shell command into the command line. This is useful for compiling after you have edited a cpp file.
- C-x 1 <CTRL>-x "one". Many shell commands and other emacs commands open a new window to show the output of the command. This will eliminate that window.

Help

- C-h t emacs tutorial
- C-h a *string* show help relevant to the *string*
- C-x 1 remove the help screen

- C-ESC-v <CTRL><ESC>v - scroll the help screen, but this doesn't work on a Windows machine.

Make and Makefiles

Non-trivial C++ applications are normally spread over many files – C++ files and header files. As the number of files grows, remembering which files need to be recompiled as you make additions and corrections becomes more of a problem. Which of your cpp files used a particular header that you just modified, possibly even indirectly? Which of your cpp files did you modify after the last debug session? UNIX provides a relatively simple mechanism to manage this complexity and, in the spirit of UNIX, a whole lot more. This mechanism is called "make". The "make" command executes a set of UNIX commands from a file. (By default it uses a file called "Makefile", but the user can change this by using the –f option on the make command line.) In addition, make will check all the dependencies you tell it about and compile any files that have been changed (are "out of date" in UNIX terms) since the last time they were compiled. The make command also allows you to execute other UNIX commands, like remove unwanted files, zip a group of files, compress files, etc. – in fact literally any command that you could type in yourself. This section will describe the ones we will use for EECS 211, but this is only a small fraction of the full capabilities of the make command.

The most common kinds of lines in a make file, aside from comment lines, are dependencies and commands. A dependency is a line that begins (in column 1) with a list of files (separated by blank if there are more than one), following by a colon followed by another list of files. The files to the left of the colon are called target files; the files to the right are the files on which the target files depend. For example, the dependency line

```
main.o: definitions.h  system_utilities.h main.cpp
```

says that the compiled version of main (main.o) depends on both header files, definitions.h and system_utilities.h, as well as the C++ file main.cpp. As the project grows and, for example, more header files are included in the main.cpp file, all the programmer need do is add the new files to the dependency; it is not necessary to remember every time you change one or more header files which ones are used in which C++ file. A typical make file has many dependency lines. A command line in a make file is simply a UNIX command. Command lines must have at least one tab character at the beginning. Comments in a make file begin with the '#' character. The format for a simple make file is a sequence of dependencies, each dependency followed by one or more command lines. For example:

```
project2:           system_utilities.o  main.o
                    g++ -Wall system_utilities.cpp main.cpp –o project2
system_utilities.o: system_utilities.h definitions.h system_utilities.cpp
                    g++ -Wall –c system_utilities.cpp
main.o:             system_utilities.h definitions.h main.cpp
```

g++ -Wall –c main.cpp –o main.o

The third group, for example says that the compiled form of the main program depends on the two header files and the cpp file; further, in order to bring main.o "up to date" the g++ compiler should be called using the –Wall option and the –o option.

"make" is a standard UNIX command and is invoked, like all other UNIX commands, by writing it as the first token on a line, possibly with other tokens representing file names, options, or other information to be used in executing the command. By default, make uses the dependencies and commands in a file (in the current directory) called either "makefile" or "Makefile". (It is common to use the second one because it stands out more when an ls command is executed.) The user can specify a different file by using the "-f" option and then giving the name of the make file to be used. In most cases the remaining tokens on the line refer to files listed on the left of a colon in a dependency. The user can reference several of these on the same command line. Here are some examples:

```
cayman :~ % make  project2
cayman :~ % make –f altmake main.o
cayman :~ % make  main.o  system_utilities.o
```

The first and third would use the file makefile or Makefile. The second would use the file altmake.

When the user issues a make command, the UNIX make utility opens the makefile and analyzes all the dependencies for the files listed on the make command line. For example, in the first example of the preceding paragraph, the make utility would check the dependency for project2, that is, would check to see if any of the files on which project2 depends has changed. This analysis is recursive, so any file on the right of the colon that is itself listed on the left of a colon in another dependency causes that other dependency to be checked and processed before the continuation of the first dependency. For any dependency that is out of date (some file on the right has changed since a file on the left was last created), the sequence of commands following the dependency is executed. Continuing with our example of "make project2", the make utility first checks the dependency for system_utilities.o. If system_utilities.o is out of date, the g++ compiler is called with the arguments shown. Similarly, make checks for main.o being out of date and regenerates it if necessary. Finally, if either one of the two .o files were out of date, the command following the project2 dependency is executed. There can be any number of commands following a dependency, and if the dependency is out of date all of the command following it are executed (unless one of them results in an error condition).

If the make command is called with no dependencies listed on the command line, the make utility assumes the user wants the first dependency in the file. It is also quite common to list a dependency with no files on the right of the colon. If the make command is issued for that dependency, the commands following the dependency line are

executed unconditionally. For example, a common operation after a coding or debugging session is to remove unwanted files. The makefile could contain the following lines.

```
cleanup:
        rm *.o
        ls *.cpp
        ls *.h
```

Executing the command "make cleanup" would (unconditionally) delete all files of type o in the current folder, then list in turn all the files of type cpp followed by all files of type h.

Comments can (and should) be embedded into a make file. The make utility ignores all characters following the character '#' until the end of the line.

It is also very common and convenient to have user-defined macros in a make file. These are variables that are defined and initialized at the beginning of the make file. They are typically given names that consist of all upper-case letters. The format of a definition is

NAME = *rest of line*

For example, we could have

```
OBJECTS   =  main.o   system_utilities.o      process_memory.o
SOURCES   =  main.cpp  system_utilities.cpp   process_memory.cpp
CFLAGS    =  -Wall
STUDENT   =  ljh
PROJECT   =  program2
HANDIN    =  $(STUDENT)-$(PROJECT).tar
```

(Note, a tar file is like a zip file.) One of these user-defined macros, or in effect variables, can by used later in the file (including in a later macro definition) by using the character '$' and parentheses. When the variable is so referenced, the text to the right of the equal sign is substituted for the variable. So, for example, the variable HANDIN has the value ljh-program2.tar because the two variables STUDENT and PROJECT get replaced by their respective values. It is common to use variables like OBJECTS and SOURCES in projects where the set of files is likely to expand. For example, in EECS 211 we add new header and source files as the project proceeds. Listing the relevant files as the values of variables like this avoids the need to search through a (sometimes long) makefile every time a new source and header file are added. For example, in the presence of the above definitions, we might include a dependency like

```
$(PROJECT):  $(OBJECTS)
        g++  $(CFLAGS) $(SOURCES) –o $(PROJECT)
```

Then the command “make project2” would check if any of the object files were out of date and regenerate the executable if they were. An interesting feature is that any of these can be overridden from the command line. For example, if you wanted to rebuild the project but not see the warning messages, you could use the command

```
cayman :~ % make project2 “CFLAGS = “
```

Simple Debugging with gdb/ddd

By now you know that even if a program compiles and links it still may not be correct. It may run to normal termination but produce the wrong results, or it may terminate abnormally ("crash"). One way to try to find the mistake is to simply look at the code. For example, if the output for some variable is not what the programmer expected, a good first step is to trace back from the output statement through all the various steps that led to the final value for that variable. A second way to help find a mistake is to insert additional output statements that print variables and other information, such as which function the output statement is in, along the way so that the programmer can see the progress of the computation and tell where it went wrong. After the mistake is fixed, these extra output statements can be removed. Both of these techniques are useful when the program is relatively small or when a small amount of new code has been added to a larger project. In many cases, however, looking at the code or intermediate steps is not easy, and a special tool called a debugger is the most productive method for finding and fixing errors.

A debugger is a tool that allows the programmer extensive control over the execution of a program and allows the programmer to stop the execution and examine the state of the executable at the point where the program was stopped. In a normal "run", the program continues execution until it terminates normally (typically by reaching the end of the main function, but there are other means for the programmer to make the program terminate) or the program crashes because of some physical error such as divide by 0 or, more commonly, attempt to access memory outside that which belongs to the program. A debugger allows the programmer to specify points, called breakpoints, at which the execution of the program should stop. Once stopped, the programmer can examine and even modify variables at that point, see how the program got to that point (examine the execution or "call" stack), continue the execution, either to the next breakpoint or by proceeding one statement at a time, step into functions, etc.

Here are some examples of how a programmer might use a debugger.

1. An output statement produces the wrong value. The programmer could select some strategic breakpoints along the way to the output statement. As execution stopped at each breakpoint the programmer examines the variables to see if the intermediate steps are ok. The programmer might continue to the next breakpoint or single-step through a critical section to see if the right branches on conditional statements or loops are being taken.
2. An output statement produces the wrong value, but there are many ways to reach that point in the program. The programmer might set a single breakpoint at the output statement. When the execution stops, the programmer examines the call stack to see how the program got to that point for this particular case of input data. The programmer might then look at the code for the functions that lead up to this breakpoint or might set breakpoints at the beginning of those functions and start the debugging over, proceeding as in example 1.

3. The program crashes. This kind of error is very often due to bad pointers, either uninitialized or NULL pointers. The UNIX system will list the address of the point in the executable program where the error occurred. The programmer can use the debugger and set a breakpoint at that address. Once the program stops, the programmer can check the call stack to see how the program got there and then either examine the code or set new breakpoints and restart the debugging.

Most modern compiler systems have associated debuggers. UNIX has the gdb, a relatively simple debugger, and gdd, a visual-style debugger, each as separate tools in the UNIX system. VisualC++ has a debugger built into the integrated VisualC++ environment along with the compiler, editor, and other tools. Although the format for performing operations and specifying things like breakpoints may be different in the various debuggers, the basic functionality is the same for all debuggers and includes:

- the ability to set and clear breakpoints;
- the ability to examine and modify variables;
- the ability to evaluate expressions;
- the ability to step through the program one statement at a time, selectively stepping over or into functions that occur in the statement;
- the ability to step out of the current function;
- the ability to continue execution to the next breakpoint or to skip over a breakpoint some number of times (useful when the breakpoint is inside a loop and you want to see what happens the last one or two times through the loop).

As with other UNIX tools, gdb is a command-line based tool. The executable file must have been built using the –g or –ggdb option. For example,

```
cayman :~ % g++ -Wall –ggdb hello.cpp –o hello
```

Then rather that executing the program hello (i.e., rather than cayman :~ % ./hello), type

```
cayman :~ % gdb
cayman :~ % gdb file hello
```

The first command starts the gdb tool; the second loads the executable file that you want to debug. Alternatively, you could combine the two into a single command:

```
cayman :~ % gdb hello
```

You can find out other command-line options by typing

```
cayman :~ % gdb -h
```

Most of these will not be useful in this course. You terminate a debugging session by typing quit.

```
cayman :~ % gdb quit
```

If the program you are debugging has not yet reached its own termination, gdb will warn you and ask if you want to continue.

One of the first operations before beginning debugging is to set breakpoints. You may set a breakpoint at the beginning of a function, at a specific line within a function, or at a particular physical address in memory (which you might obtain from the error message when your program crashed). Examples are

```
cayman :~ % gdb break main
```

```
cayman :~ % gdb break student::print();
cayman :~ % gdb break student.cpp:27
cayman :~ % gdb break 0x8048e30
```

You may also list the breakpoints by typing the command

```
cayman :~ % gdb info break
```

which will produce a list of the breakpoints, each with its own number and other information. (See further down for more information about info.) You can remove breakpoints when you no longer need them by typing either of the commands

```
cayman :~ % gdb clear function
cayman :~ % gdb delete breakpoint-number
```

Once the breakpoints have been set you start the debugging of the program. The run command starts at the beginning of the program and continues until a breakpoint or termination. Note, if you had already been debugging, this command will ask you if you want to start over. (You could also start the debugging session by issuing the command to "step into" main. See later in this paragraph.) When a breakpoint is reached, the various options for continuing the execution are:

- continue – a resume execution;
- run – start over;
- next – execute the next instruction;
- next n – execute the next n instructions;
- step – like next except step into any functions in that line;
- finish – step out of the current function;
- backtrace – show the call stack for the current position in the program.

Of course, normally you want to look at things when you reach a breakpoint. The print command is used for this purpose. You can print individual variables or array elements or even entire arrays that are within the scope of the line where the debugger is currently stopped. For example, assuming "process" is a class with a data member number_of_files,

```
cayman :~ % gdb print i
cayman :~ % gdb print process->number_of_files
cayman :~ % gdb print process_list[2]
cayman :~ % gdb print process_list@4
```

You can also arrange to have variables displayed automatically whenever a breakpoint is reached by using the display command; see the manual or the appendix for details.

Very often you discover that some variable has the wrong value. Rather than quitting and having to start over after you fix the code, thus wasting all the time spent in the session so far, it is useful to be able to "fix" the value of the variable so you can continue debugging and continue checking the remainder of the program. You can do this by using the variable option of the set command:

```
cayman :~ % gdb set variable variable = expression
```

(Note, the set command has many options, so be sure to include variable on the command line.) For example, suppose you used to print to find out that the value of the argument a

in a function you just entered was wrong and you wanted to continue debugging with a reasonable value. You could give the command

```
cayman :~ % gdb set variable a = 5
```

The expression can involve variables within the current scope of execution, for example,

```
cayman :~ % gdb set variable a = a+1
```

The info command provides a variety of useful information, such as what variables are in the current scope, what are the global variable, what breakpoints are currently active, etc. You can type info on then command line and get a list of the command-line options.

```
cayman :~ % gdb info
```

We have already seen one example of this command to get a list of the breakpoints. Two very useful command options are locals and args, which list the local variables (excluding the function arguments) and the arguments of the current function respectively.

```
cayman :~ % gdb info locals
cayman :~ % gdb info args
```

See the manual or the appendix for details.

The help command provides information about the gdb commands at the level specified on the command line. For example,

```
cayman :~ % gdb help
```

provides a list of the different classes of commands that that are relevant at the top level of the gdb command line – e.g., data for command that have to do with data (like print) or breakpoints for commands that have to do with breakpoints. Typing help and a command class gives a list of the commands in that class:

```
cayman :~ % gdb help data
```

Typing help and a command, e.g. set, gives information about that command:

```
cayman :~ % gdb help set
```

You can continue to drill down to get the details as needed, for example,

```
cayman :~ % gdb help set variable
```

Most gdb commands have abbreviations to reduce the amount of typing required. In most cases the abbreviation is the first letter of the command (or first few letters if two commands begin with the same letter) or the first letters of the two words of a compound word command (e.g., bp for breakpoint).

Appendix 3 provides a summary of the most commonly-used commands for gdb.

The ddd debugger is a Windows-style front end for gdb and has many aspect similar to the Visual C++ integrated debugger. The complete ddd user's manual is posted on the Blackboard site. Section 1 – A Sample DDD Session – provides a good overview of how to use the ddd debugger. The remainder of the manual provides details of the individual features and operations. Most of that material concerns either advanced options or formatting options. A close examination of Section 1 should be sufficient for most of the debugging you will do in this class. NOTE: The ddd debugger requires a windows-based access to the UNIX system; most SSH programs do not have the display capabilities necessary to run ddd.

Appendix I Commonly Used Unix Commands

Here is a list of the top-level commands you will use most often in this course. It is provided as a quick reference in the early weeks. Most likely after a week or two you will be so familiar that you will no longer need this list. Some sample uses are also shown in lieu of a listing of options for each individual command.

Help command

- man display the manual for a given command.
  e.g. man ls

File and directory commands

- ls list the files and directories in the current working directory
  e.g. ls *.cpp
- pwd print the name of the current working directory
- cd change the current working directory
  e.g. cd project3
  e.g. cd .. (go up one level in the directory structure)
- mkdir make a new directory
- rmdir remove (i.e., delete) a directory
  e.g. mkdir project4
  rmdir project2
- rm remove (i.e., delete) a file
- cp copy a file to another place
  e.g. cp main.cpp main.tmp
  e.g. cp main.cpp project4 (project 4 is a directory)
- cat display a file
- less display a file one screen at a time
- head display the first 10 lines of a file
- tail display the last 15 lines of a file
  e.g. cat definitions.h
  head main.cpp
- grep find occurrences of a word or phrase in a file
  e.g. grep MAX_FILES main.cpp

Compiling and debugging

- g++ compile a file or group of files
  e.g. g++ -Wall main.cpp system_utilities.cpp –o project2
- make invoke the make file to do a set of jobs

- gdb invoke the debugger
- *./file* execute the program in *file*

Editor

- emacs invoke the emacs line editor
  e.g. emacs system_utilities.cpp

Appendix II Commonly Used EMACS Commands

Remember, C-*letter* means hold the "Cntrl" key and then press *letter*, while ESC-*letter* means press ESC and *letter* in sequence.

Quit emacs:

- C-x C-c quit completely
- C-z suspend emacs

Basic file manipulation:

- C-x C-s save
- C-x i insert another file into this file at the cursor position
- C-x C-w *file* write the edit buffer to the file

Navigation in text

- Use the four arrows (up/down/left/right)
- ESC-< go to the beginning of the file
- ESC-> go to the end
- C-v or PageDown scroll down one screen
- ESC-v or PageUp scroll up one screen

Undo

- C-x u undo the last command or keystroke
- DEL delete text backwards
- C-d delete text forwards.

(NOTE: The "Delete" key inserts a tilde character "~" but does not delete text.)

Search

- C-s *string* begin dynamic search forward.
- C-r *string* begin dynamic search backwards.
- <ENTR> terminate the search
- C-s move to next occurrence of the search string
- C-r move to next occurrence of the search string
- ESC-p/ESC-n change to next/previous search string

Copy and Paste

- mouse mark region, then copy/paste
- C-*space* mark beginning of text block for a region
- C-x r s *reg* mark end of the text region and save in register *reg*
- C-x r i *reg* insert register *reg* text at cursor position

Shell

- ESC-! execute a shell command
- C-x 1 close the window opened by the last shell command (Note,

the last key for this command is “one”, not “el”.)

Help

- C-h t emacs tutorial
- C-h a *string* show help relevant to the *string*
- C-x 1 remove the help screen
- C-ESC-v <CTRL><ESC>v - scroll the help screen
- C-h a *string* show command matching *string*

Appendix III Commonly Used gdb Commands

Remember to use the –ggdb option when you build the project.

Start the debugger

- gdb starts gdb without loading the program
- gdb *file* starts gdb and loads *file*

Quit the debugger

- quit

Help

- gdb –h display command line options
- help display a list of the gdb commands
- help *cmd* display information about *cmd*

You can continue to drill down to get the details as needed, for example,

help set variable

Breakpoints

- break set a breakpoint at a function, file line, address, etc.
  e.g. break main
  break system_utilities.cpp:20
- clear *function* clear the breakpoint at the indicated function
- delete *n* delete breakpoint number *n*
- info break list the current breakpoints
- continue resume execution
- run start over
- next execute the next instruction
- next *n* execute the next n instructions
- step execute the next instruction, stepping into any function
- finish step out of the current function
- backtrace show the call stack for the current position in the program

Displaying and changing variables

- print *var* show the value of *var*
- display *var* show the value of *var* each time execution stops
- set *var exp* change the value of *var* to *exp*

Information about debugger attributes

- info display a list of options for the info command
- info *item* display information about *item*

  e.g. info variables
  info locals