Laboratory Manual

CISC 105
General Computer Science

Department of Computer and Information Sciences
University of Delaware

September 2003 through August 2004

Contents

Part I

Course Information

Chapter 0

Course Information & Guidelines

0.1 Professor and Teaching Assistant (TA)

You will receive the following instructor and TA contact information in lecture or lab during the first week.

Professor:	TA:
Office:	Office:
Phone:	Phone:
Hours:	Hours:
Email:	Email:

Note: Your first reading assignment - after you read this chapter of the lab manual, read chapters one and two (pages 1-28) in your Unix textbook, and chapter one in your C programming textbook.

0.2 Overview

CISC105: General Computer Science is primarily an *introductory computer programming* course designed for two groups of students: (1) mathematically-oriented non-CISC majors, and (2) CISC or potential CISC minors/majors without significant programming experience. CISC105 assumes *no* prior programming knowledge.

The following are *good* reasons to take CISC105:

- You are a CISC major[1] without significant prior computer programming experience which is prerequisite for CISC 181: Intro to Computer Science.
- You are thinking of becoming a CISC minor/major and need CISC105 and CISC181 before your Change of Minor/Major Request will be considered by the CISC Department.
- You want to learn about:
  - Design and implementation of small to medium sized programs in C, including compilation, testing, debugging and execution of these programs.
  - Principles of software engineering including naming conventions, program structure, and methodology.
  - Familiarity with the Unix Operating System for a variety of purposes including the creation, editing, transfer and printing of files.
  - Use of the "X" window-based user interface.
  - Use of local network and Internet resources.
  - Basic ideas of computer science including program and data structures, file management, computer hardware components, and algorithm development and analysis.

The following are *bad* reasons to take CISC105:

- You could not get into CISC101. (Try ACCT160, FREC135, EGTE111, HPER276.)
- You need a group D requirement, and want to learn about computers.
- You need a course that will not require much time outside of class.

CISC105 is *not* a course in word processing. CISC105 is *not* a course that teaches canned commercial software packages, such as EXCEL spreadsheets, ACCESS databases, and POWERPOINT presentations. General computing concepts, terminology, and popular office software is covered in CISC101: Computers and Information Systems.

CISC105 emphasizes programming. Programming is a difficult and time-consuming task.

0.3 Class Meetings and Attendance

Each week consists of two lecture classes and one lab class. Students are required to attend all lectures and lab classes. **Warning: Poor attendance almost always results in poor performance.**

0.4 Texts and Readings

- *C How to Program, 4th ed* by Deitel and Deitel 2003 Prentice Hall
- *CISC105 Laboratory Manual*
- *Unix Unbounded, A Beginning Approach, 4th ed.* by Afzal 2002 Prentice Hall

[1] also CPEG, ELEG, MATH

0.5 Programming Environment and Computer Usage

The C language is available on a group of computers named Strauss, Bach, Brahms, and Chopin. They are referred to as the "Composer" machines. You will primarily use Strauss for all programming in CISC105. Strauss is a multiprocessor computer manufactured by Sun Microsystems running the Solaris Unix operating system. Primary access to Strauss is obtained via X-terminals located at various computing sites on the Newark campus and elsewhere. Additionally, dial-up access is available as described in Appendix C of the Intro to Unix handbook. For information about these computers and other University computing resources, contact the Computer Network Services (CNS) Consulting Desk, 002 Smith Hall, 831-1205.

Note: You cannot do your C work on Copland. You **MUST** do your work while logged into strauss. However, you may log into strauss from your computer, via Exceed, or a secure shell connection (ssh).

0.6 Obtaining an Account - What YOU Must do NOW

Note: If you are officially registered for this course, you will have an account set up for you on strauss.

To participate in the first lab class, you *must* have a valid university account. If you do not, then you *must* :

1. Go to one of the computing sites or log on from any terminal at which you can obtain a strauss window, and take the Electronic Community Citizenship Examination (ECCE) interactively. You can obtain the instructions for the ECCE at the following URL: `http://www.udel.edu/help/` You need to take the ECCE repeatedly until you pass it.

2. Immediately after you pass the ECCE, go tothe same URL: `http://www.udel.edu/help/` scroll to the bottom, and follow the instructions for activating your computer account. Your account will be activated overnight after you obtain a password. To insure that you can get onto your account during lab class, you MUST pass the ECCE and obtain your password by the day BEFORE your lab class.

0.7 Requirements and Grading

In determining your final grade, exams and assignments are weighted as follows:

```
20%        Hour Exams
```

```
 30%       Final Exam
 20%       Labs/Homework
 30%       Programming Assignments
----
100%
```

However, your course average may not be more than 10 (ten) points higher than your exam average.

Special considerations may have a positive effect on final grades. These include class participation, a pattern of grade improvement, an isolated bad grade, etc. Such considerations will never have a negative effect on the final grade.

Occasionally, mistakes occur in grading. After a graded item is returned, students have two weeks to submit a request to have it regraded. These requests should be submitted directly to the TA. After two weeks, no requests for grade changes will be considered. The request for regrading should include the original exam or assignment, and the reason that regrading is requested.

0.8 Graded Assignments

There are three categories of assignments: Labs, Homeworks, and Programming. Labs and Programming Assignments are described later in this *Laboratory Manual*. Homework assignments will be made available during the semester.

0.8.1 Lab Assignments

Lab assignments will be started (and in some cases completed) during the scheduled lab classes. Normally each lab assignment is due by the **beginning of your next lab class**. Most labs involve writing or modifying C programs. Each lab focuses on a particular concept discussed in the text or in lecture, and helps you to complete the programming assignments.

0.8.2 Homework Assignments

Homeworks normally consist of pencil-and-paper exercises designed to give you practice in applying information covered in the textbook and lectures. Homeworks usually do not require the use of a computer, although students can often check their answers on the computer. Homeworks will be handed out with the lab material, and are due at the **beginning of your next lab class**.

0.8.3 Programming Assignments

Programming assignments consist of applying several of the concepts from several labs and lectures. Students must write their own computer programs that need to be compiled, tested, and debugged on the Strauss computer. There will be at least 3 programming assignments.

You will be given about 2-3 weeks to complete each programming assignment. Program grades are based on: program design, correct execution, proper program comments, properly formatted output, and proper program style. While working on a program, you may consult with the TA, professor, classmates, friends, etc., but the actual programming that you hand in *must* be your *own* work.

0.8.4 Extra Credit

Some assignments may include an opportunity for extra credit. Extra credit is optional, i.e., students can earn an A grade in the course without doing any extra credit. Extra credit on programming assignments provides students with an opportunity to gain greater programming skill and to learn more advanced material while at the same time improving their course grade. Current and potential Computer Science majors are encouraged to do the extra credit assignments to facilitate transition into the higher level Computer Science courses.

0.8.5 Late Submission of Assignments

An assignment's due date will be clearly specified when the assignment is made. Do *not* miss class or lab in order to complete an assignment. Late assignments will be penalized unless an extension is granted by the Professor. Only the Professor (not the TA) may grant an extension for an assignment.

The philosophy on late assignments is: (1) Everyone should try his/her best to complete all assignments by the due date. (2) People who work conscientiously to meet deadlines should be rewarded for their promptness and sacrifice of sleep. Thus, allowing others to hand in late assignments without some penalty is unfair to these people. However, there are various circumstances that may prevent a student from completing an assignment on time. Allowing no late assignments would not give students much incentive to eventually complete their work, which is a major source of learning. Thus, I believe that late assignments are better than no assignment.

Late assignments will be penalized 5% per 24 hour period or fraction thereof (not including weekends) up to maximum 25% penalty. For example, if your lab class meets Thursday at 10am, and you submit the assignment late on Monday at 11am, then the penalty is 10%. Late assignments will be accepted with penalty up to one week following the due

date. **Assignments submitted more than one week late without an approved extension will not be accepted.**

It is best to submit late assignments directly to the Professor's mailbox in 103 Smith Hall, or the Professor's office. An assignment is considered turned in when it is physically received by the TA or professor or placed in their mailbox, not when it is printed by the computer. Please write the date and time on the late assignment just prior to submitting it.

Note: Extra credit work cannot be handed in late.

0.9 Exams

There will be one or two exams during the semester, and a final exam. Each exam is closed book and in-class. The final exam is comprehensive.

If an exam is missed because of an absence that **has been excused** by the Professor, arrangements will be made either to take a make-up or to increase the weighting of the other exam, according to the Professor's choice. If the absence is not excused, it cannot be taken later, and a score of 0 will be included in the computation of the final grade.

0.10 Readings

Be sure to keep up with the readings - be especially deligent with the readings for labs - these should be completed before arrive for your weekly lab. The labs assume you have done the reading before hand.

0.11 Academic Honesty

For many students, this is the first time you will be programming a computer and the first time you will be "debugging" computer programs. Hence, you need to learn what is allowed behavior and what is not. When your program does not work, the first thing to do is use your book and notes to try and figure out the problem yourself. The second and third things to do are to try to figure out the problem yourself! At that point, you may ask for the assistance of a consultant, TA, Professor, classmate or friend to help you understand the specific problem.

You may also discuss in *general* terms the *general* approach to solving a programming problem. Once the discussion gets down to specific programming issues such as names and types of variables to use, control structures such as loops, if-then-else statements, you must end any collaboration.

Specifically, you may *not*:

- Compare answers to any assignment before it has been turned in.
- Supervise a classmate typing in a program, or have a classmate supervise you typing in a program
- Copy, or allow another student to copy, a computer file that contains another student's assignment, and submit it, in part or in its entirety, as your own.
- Work together on an assignment, sharing the computer files and programs involved, and then submit copies of the assignment as one's own individual work.
- Edit a script file, and then submit it as an original transcript of your computer session.

Any evidence of performing any form of academic misconduct will be appropriately handled as stated in the Official Student Handbook of the University of Delaware. If you are in doubt whether or not a behavior is permitted, then ask the Professor or TA beforehand. If you are having difficulty with the course, then see the Professor or TAs for help.

Part II

Laboratory Assignments

Lab Assignments

Lab assignments will be started (and in some cases completed) during the scheduled lab classes. Normally each lab assignment is due by the **BEGINNING of your next lab class**. Most labs involve writing or modifying C programs. Each lab focuses on a particular concept discussed in the text or in lecture, and helps you to complete the programming assignments.

Homework Assignments

Homeworks normally consist of pencil-and-paper exercises designed to give you practice in applying information covered in the textbook and lectures. Homeworks usually do not require the use of a computer, although students can often check their answers on the computer. Homeworks will be handed out with the lab material, and are due at the **BEGINNING of your next lab class**.

What Computer Do I Use?

All program projects and lab work requiring an online program must be done on a Unix machine - strauss for this course. You access strauss remotely - either from an X-terminal or PC in a public lab, or from your personal PC. Use either a telnet session, or X-windows. You must learn Unix, and use the Unix C compiler for this course.

Chapter 1

Lab - Logon, setup, unix, and program edit-compile-execute

1.1 Goals

This lab is an exercise to familiarize you with the University of Delaware computing environment. You will learn how to:

- Login and logout of the system.
- Improve the security of your password.
- Change your default project.
- Access computer files for CISC 105.
- Use the X–terminal interface to communicate with the computer named `strauss`.
- Become familiar with the `vi` text editor.
- Enter, compile and execute a C program using a C compiler named `cc`.
- Create a script file which records your computer session for grading.

1.2 Reference Materials

Afzal, Chapters 3-4, pp 31-82.

1.3 Step 1: Getting an account and password

Every user of the University computer system must have three things: a login number or name, a password, and a project number. Only students officially registered for the class are authorized for this class's project. You should already have a login number or login name and a password from previous classes or from having established an email account, or through freshman orientation.. If not, see syllabus.

1.4 Step 2: Logging in

A login window should already be on your screen when you sit down at your X-terminal.

- Type in your user name or number in the space provided after `Login:`.
- Press the RETURN (or ENTER) key.
- Type in your password at the password prompt and press RETURN.
- If you receive an `incorrect login` message, your number/name and password combination is invalid. Perhaps you made a typo; try again. If you still cannot login, your account is not yet active.

Congratulations! You are now logged into the machine named Strauss. You should see the "C shell" prompt, >. This means that the computer is waiting for you to type commands for it to execute. Whenever you type commands to the computer, you will need to press the RETURN or ENTER key after each command to send it to the computer. When class is over or whenever you need to log out, skip to the last section.

1.5 Step 3: Changing your password

Your password is your security against unauthorized users tampering with your files. Your password should:

- Be 5-8 characters long.
- Not be a word that is found in the dictionary.
- Contain both upper case and lower case letters.
- Contain at least one non-alphabetic and one alphabetic character.
- Be changed whenever you think someone has found out your password.

- Not be given to anyone including your friends.

To change your password, type the command `passwd`. Follow the instructions that are displayed. When you change your password, the change is not immediate; it becomes effective sometime within 24 hours. The new password will be recognized by all of the composer computers. Do not forget your password. It is encrypted within the computer and no one can access it for you. If you forget it, go to Smith 002A for help. You can change your password at any time by typing the passwd command.

1.6 Step 4: Creating a login name

Your initial login is a number that will uniquely identify you for as long as you are a computer user at the University of Delaware. However it is more convenient to be known by a name rather than by a number, so you can specify a login name for yourself. Note, however, that YOU CAN ONLY GIVE YOURSELF A USERNAME ONCE, and it cannot be changed thereafter, so choose a name that you can live with for a long time. Most people simply use their last name as their login name. Using just your last name is my recommendation since other people often have to guess your user name, for example, when sending you an email message. The next most common approach is first letter of first name followed by last name, e.g., `tsmith`. Others choose more fanciful names such as `starbaby`, `monsterman`. The choice is yours; just remember, your login name is like a tattoo. Once selected, you cannot change it.

To create a name, type the command `username` (and as always, hit RETURN or ENTER.) You will be prompted to give the username that you want. If you have been getting email at your account which still has a user number, then you will now get email addressed to your user name as well as your user number.

1.7 Step 5: Changing your default project

When you use a computer, you are in effect renting time on it, the cost of which is charged to a project number. At the University of Delaware, you may be authorized to use several projects. For example, most students are authorized to use an email account which has project number 4000. Once officially registered you will be authorized to use your class project for doing computer work for CISC105. You can find out the project number for this course by entering the command `chdgrp` at the unix prompt - note the four digit number with `CSIC 105` annotation. If you don't see your project listed, get help from your TA.

Whenever you log on, the computer time that you use is automatically charged to a project called your default project. If you have an email account, probably project 4000 is your default. Check to see what project numbers you are authorized to charge your time against

and which of these is your default project by typing the Unix command `chdgrp`. Here is an example execution:

```
strauss.udel.edu% chdgrp
Project    Title                  Remaining      Valid on hosts
 1286   RESEARCH                     300.00       strauss chopin bach ravel
 2182   CISC105                     6000.00       strauss chopin bach ravel
 3558   CISC672-10                   100.00       strauss chopin bach ravel
 4000   U. OF D. E-MAIL               50.00       strauss chopin bach ravel

default group is currently 4000
```

In the above case, the user is authorized to use any of four projects, and the user's default project when they logon is currently 4000.

In this example, project number `2182` is the CISC105 class project—your actual project number will be different. It is important that you use your CISC105 project when working on CISC105 assignments. There are two ways to do this; the second way is recommended: (We'll use `2182` as an example - make sure you use you actual project number when entring the commands.)

1. Whenever you want to do CISC105 work, type the command `newgrp 2182`. This will switch you to project `2182` until you type `exit`. You must do this every time you want to do CISC105 work.
2. Type the command `chdgrp 2182`. This will change the default project so that you will be working on project `2182` automatically each time you log on in the future. Use the command `newgrp` when you want to work on non- CISC105 assignments. Note: it takes up to 24 hours for your old default project to be replaced by the new one. An example is shown below.

   ```
   strauss.udel.edu% chdgrp 2182
   Changing default project for smith
   password:
   The changes will be made within 24 hours
   ```

1.8 Step 6: Accessing CISC105 Files

It will be necessary to access files for CISC 105. The easiest way to access files for labs and program assignments is to use a web browser(eg. Netscape). Simply go to `http://www.udel.edu/CIS/105` - all files needed for labs and programs are there.

However, if you are working at a PC (eg. in your dorm room), then a browser is not the best way to access files for CISC 105. Instead you will have to link to them using unix. The Unix directory (i.e., path name) for these files is `/www/htdocs/CIS/105`.

1.9 Step 7: Creating and Editing a File on Unix

Note: Be sure to study chapter 4 of Afzul before beginning this section. You will be using the *vi* text editor - this is your first major challenge - to learn this editor. Do all of the practice edit sessions in Chapter 4 of Afzul - practice is the only way to learn an editor!

Information is stored in the computer in files. You can think of a computer file as just a manila file folder, which contains information such as a computer program, a letter, data for a computer program, a term paper, a resume, or any kind of information that can be stored as a sequence of visible characters such as a-z, A-Z, 0-9, and punctuation. An editor is a program that helps you create and store a text file in a computer and then to later modify it.

Computers usually have several editors and you must learn one of them. One of the easiest to learn at Delaware is `vi`. One of the most powerful (and complex!) is `emacs`. Both are part of the standard Unix operating system which controls the operation of each of the composers. You are free to use any editor. However, `vi` is strongly recommended for those not already familiar with an existing Unix editor. Only `vi` is discussed here.

Start the `vi` editor (by entering the command `vi lab1.c`). Enter *insert mode* by typeing an "i" (this letter will not be visible on the screen when you type it). Now, enter the following C program **exactly** as shown with one exception; modify the comments at the beginning to reflect your name, class section, and today's date. Do not add or delete any blanks. You do NOT have to type the exact number of stars. Call the file `lab1.c`. Once you have finished entering the file, type the escape key to leave insert mode - then you can use arrow keys to move arround in the file. Fix and mistakes you made, then exit the editor by typing `:wq` - meaning write the file and quit.

```
/*****************************************************************
 * Programmer:                                                   *
 * Course: CISC 105                                              *
 * Section:                                                      *
 * Time:                                                         *
 * Date:                                                         *
 * Title: Introductory Example: Compute n!                       *
 *                                                               *
 * This program inputs an integer value in the range 1-15 and computes *
 * and outputs the 'factorial' of that value.  By definition, the *
 * factorial of an integer n (denoted as n!) is:                 *
 *     n! = n * (n-1) * (n-2) * (n-3) * ... * 3 * 2 * 1          *
 *             e.g., 5! = 120 = 5 * 4 * 3 * 2 * 1                *
 *                                                               *
 *****************************************************************
 */

/* include the standard input/output library routines */
#include <stdio.h>

int main() {
   /* Declaration and description of variables */
   int n;      /*  value inputed by the user   */
   int nfact;  /*  computed n! value           */

   /* get input value from the user */
   printf("Enter a value between 1 and 15 (inclusive):  ");
   scanf("%d", &n);

   /* test for valid input */
   if ((n <= 0) || (n >= 16)) {
      /* input was not valid */
      printf("You have entered an invalid value.  Goodbye!\n");
      exit();
   }

   /* compute n! */
   nfact = factorial(n);

   /* print the results */
   printf("the factorial of %d is %d\n\n", n, nfact);
}

/*****************************************************************
 *              Function: factorial                              *
 *                                                               *
 * Purpose:  recursively compute the factorial of an integer     *
 *****************************************************************
 */
int factorial(int i) {
   if (i <= 1)
```

```
        return (1);
    else
        return (i * factorial(i-1));
}
```

You can move around in `vi` by using the arrow keys. If you make an error in typing the program text, consult Afzal to learn how to correct your mistakes.

Once you have typed in the C program (paying close attention to the punctuation), save and exit from*vi*.

You can verify that your file has been saved by typing `ls -l lab1.c` at the command prompt. The next time you type `vi lab1.c`, you will be editing the most recent copy that you saved.

1.10 Step 8: Compiling a C program

You have just finished creating a computer file that contains a real live C program. Before you can see if your C program works by executing it, your C program must be translated into a format that the computer can understand directly. We call the computer understandable file an "executable file". This translation process is known as **compiling** and is performed by another program called a **C compiler**. There are several C compilers available. The one we use is the Sun 4.2 C compiler. Type the command `cc lab1.c` to invoke (execute) this compiler.

If you made any mistakes typing in the given program, the compiler will now give you error messages. Use `vi` to go back and edit your program. The error messages will not mean much at this point since you do not know the C language. Do your best. Go to the indicated line in error and look at the lines before and after this line for something that was typed incorrectly. An error message may say the error is in line 20, but the real error was omitting a ";" in line 15. This process is called debugging.

After you have corrected some errors, go back and recompile the program by exiting vi and typing `cc lab1.c` again at the Unix prompt. Continue this cycle until compilation results in no error messages. When compilation completes with no error messages, type `ls` to list the names of all of the files in your directory. The list should include both `lab1.c` and `a.out`. The `a.out` file is the executable file created by the C compiler.

1.11 Step 9: Executing a C program

Once there are no compilation errors, it is time to execute your program. Compilation identified *syntax* errors such as spelling mistakes and missing punctuation. Execution requires valid *semantics*. If you try to divide by 0 (which is not allowed!), you will not get a

compilation error message. Instead you will get an execution error message. Even worse, if you type "+" when you want the computer to do subtraction, you will get NO error message; the program will just generate incorrect results!

To execute your compiled program, enter the command `a.out` at the prompt. The file `a.out` is the automatic (i.e., default) file name the compiler gives to the executable version of your program. You cannot understand the insides of the file `a.out`. Do not try to print, edit, cat, or otherwise look at an executable file.

Hopefully you will now see a valid execution of the `factorial` program. That is, you should see the factorial displayed on your screen for the number you entered. If not, then continue debugging.

Question: Take a careful look at the output from this program—do you see anything that looks incorrect? If so, can you explain how it happened?

1.12 Step 10: Scripting Your Session for Grading

Note: Do not enter the *vi* editor during a scripted session.

Unix provides a way for you to "capture" all of the information that appears on your terminal screen during the time that you are logged in, and to save it in a file. By typing the command `script file.scr`, from that moment on, the computer will save whatever you type and whatever the computer displays on the screen in the file called `file.scr` until you type `exit`. After typing `exit`, no more information from the screen is saved in the file.

This capability allows you to show others what you actually did while you were working on the computer, without them having to look over your shoulder. In particular, it allows the TA or Professor to see what you actually did, and grade you on your work. The process is two steps: (1) create a script file that contains what has been displayed on the screen. (2) print out the script file and hand it in for grading.

For this lab, you should execute the following commands:

```
% script lab1.scr
% cat lab1.c
% rm a.out
% cc lab1.c
% a.out
% exit
```

The first line results in the following actions:

1. Unix creates a file named "lab1.scr". WARNING! If "lab1.scr" already exists, Unix ERASES it; BE CAREFUL! The filename that you use should NOT end in `.c` If you

type, "script lab1.c", you will ERASE whatever was in the file lab1.c, which WAS your C program!

2. In addition to showing you the info on the screen, Unix saves everything that appears on the screen in the file "lab1.scr".

3. Unix watches your input, looking for you to type "exit" after the prompt. When you do so, Unix stops storing information in "lab1.scr". DO NOT FORGET TO TYPE "exit"; otherwise your script file will be lost.

The second line will display your contents of file lab1.c on the screen. This will also get saved in the file lab1.scr because you are in script mode. The third line removes any a.out file you may have in your directory before compiling and creating a new one. The fourth line compiles your program, the fifth line executes it and displays any output on the screen, and the sixth line exits the script mode. At this point, you can view your script file, lab1.scr, by typing `more lab1.scr`. If the whole file is not displayed, hit RETURN or ENTER to see the remainder of the lab1.scr file.

IMPORTANT WARNING:

You are **NEVER** permitted to edit a script file in any way before submitting it. **NEVER** means **NEVER**; not even to change your name's middle initial. Editing a script file and then submitting it is considered academic dishonesty.

1.13 Step 11: Printing out your work and what to hand in

After exiting "script", you can print a copy of your script file using the command `qpr`.

To print your script file using the Willard Hall laser printer, type the following command:

```
qpr -q whlps lab1.scr
```

To print your script file using the Pearson Hall laser printer, type the following command:

```
qpr -q prsps lab1.scr
```

To print your file in other buildings, you would substitute the name of the printer in that building (usually written somewhere on the printer itself) for `whlps` in the above command. For example, `qpr -q smips lab1.scr` uses the printer in Smith Hall basement.

You can tell if your output is waiting to be printed or has finished printing at the Willard Hall printer by entering the `qstat -q whlps` command soon after you execute the `qpr` command.

Go ahead and print out your lab1.scr file. This is what you should hand in to the TA for grading of lab 1.

1.14 Step 12: Logging out

Pull down the Root menu and hit "quit and logout". You have successfully exited the system when the original login window appears. **Never leave a terminal without logging out; otherwise the next user will have access to all your files!** If you sit down at a terminal where the previous person left and did not log out, simply log that person out for her/him. Do not look around at that person's files! Even looking without touching is an invasion of privacy, and can get you in trouble.

1.15 Step 13: Follow Up

Do the `terminal Session` in Afzul on page 52 - and be sure to script the session for grading.

1.16 What to Hand In

Rememebr that lab work is due at the next lab session. Hand in all of the scripts files from this lab - stapled as one package with your name and section number on the front page.

Chapter 2

Lab - Input/Output, Files, Email

2.1 Goals

This lab is an exercise to familiarize you with the following:

- Sending and reading email.
- Some Unix commands for working with files.
- Interactive input and output with C and Unix.
- Using data files for input and output.
- More work with *vi*.

Your primary task this week is to become competent at using the *vi* text editor to create and edit files.

2.2 Reference Materials

Afzal, Chapters 4 and 5

Chapters 1 and 2 of textbook (Deitel).

2.3 Step 1: Email

Note: Your TA will tell you which email address to use for this part.

For the first step of this lab assignment, enter the `pine` unix command and **send a mail message to the TA** with answers to the following questions. IMPORTANT: In the Subject line of your mail message, include the keyword CISC105.

1. What year are you?
2. Have you used computers before, say for word processing? If so, indicate how?
3. Do you expect to take another computer science course after CISC105?
4. What is your major? Does your major require you to take CISC105?
5. If you are not a CIS major, are you considering CIS as a major? as a minor?
6. List at least one thing that you hope to learn in this course.

2.4 Step 2: Working with Files

The operating system that coordinates all of the activities and files of the strauss computer is called Unix. It is a popular operating system used on many different computer systems. The Unix operating system organizes all of the files of all of the users of the computer using directories. Each user of the computer has a `home directory` which is where you start each time you login. Type `ls` to see all of the files that you currently have stored on your home directory. All of the directories of files on strauss are organized in the form of a tree.

To get more familiar with Unix commands try each of the following, and note the effect.

```
% ls
% ls *.c
% ls -l
% ls mail
% ls junk
% ls -al
% man ls
```

Read the description of the commands: `cp, rm, mv, cat, more, and qpr` in the Appendix and in Afzul. These are commands you will often use when working with files.

2.5 Step 3: Interactive Mode in C Programs

In C, the `printf` statement causes the contents of the printf to be displayed or output to the screen for the user to view. The `scanf` statement causes the C program to pause execution

at that point in the program execution, and wait for the user to type the proper input. When the user hits the RETURN or ENTER key, execution continues and the program uses the input from the user in a way depending on the contents of the scanf statement.

For this lab, perform the following to learn about interactive input/output in C:

1. Use *vi* to enter the following program into file called `lab2.c` and save it into your own home directory.

```
/*************************************************************************
 * Programmer:                                                           *
 * Course: CISC105                                                       *
 * Section:                                                              *
 * Lab Time:                                                             *
 * File: lab2.c                                                          *
 * Date:                                                                 *
 * Title: Input/Output in C                                              *
 *                                                                       *
 * This program performs a simple computation of a customer's bill.      *
 *                                                                       *
 *************************************************************************
 */

#include <stdio.h>

main() {
   /*declaration of variables*/
   int   num_pears;            /*number of pears bought       */
   int   num_oranges;          /*number of oranges bought     */
   float price_pear = .55;     /*price of 1 pear              */
   float price_orange = .40;   /*price of 1 orange            */
   float subtotal;             /*subtotal of bill without tax */
   float totalbill;            /*total of bill including tax  */

   /*determine the quantity of goods bought by this customer*/
   num_pears = 7;
   num_oranges = 5;

   /*calculate the bill subtotal and total*/
   subtotal = (num_pears * price_pear) + (num_oranges * price_orange);
   totalbill = (.07 * subtotal) + subtotal;

   /*print out the total*/
   /*printf("Total is: $%5.2f\n",totalbill);*/
}
```

2. Make sure that you put your name and the date in the comments at the beginning of the program.

3. Examine the program, and make sure that you understand what it is doing. Compile

the program by typing `cc lab2.c`. Execute the program by typing `a.out`. Surprise!

4. Modify the program so that the total is displayed on the screen when you execute it, recompile the program, and execute it again. You should get a total printed out that looks like:

```
Total is:  $ 6.26
```

5. Add statements to the program so that the output looks like:

```
The number of pears bought is 7.
The number of oranges bought is 5.
The subtotal of bill is $ 5.85
Total including 7% tax is: $ 6.26
```

The statements that are added should print out the current values of the appropriate variables to get the numbers to print out. That is, there should be no statements that look like: printf("The number of pears bought is 7."); The printf statements should not explicitly include the number to be printed such as the 7 above. Recompile and execute your program to check that it does indeed display this on the screen.

6. Now, replace the two assignment statements to num_pears and num_oranges by printf and scanf statements so that the program pauses execution to prompt the user for these values. The prompt should include a question to the user such as "How many pears do you want to buy?" Recompile your program and execute it. When your program displays the prompt to the user, be sure to type in the same numbers as the original program, 7 and 5, so you can check your results. This is called interactive input/output because the user is interacting with the program as it executes in order to give the program the data, and for the program to display the results on the screen!

Note: Don't forget that ampersands are needed in front of variables in `scanf`, but **NOT** in `printf`.

7. Note: **Before** you do this step - go to Appendix A and read the description of the Unix `script` command. To demonstrate that you have mastered interactive input/output, enter script mode, and cat your final version of the lab2.c file, remove the file a.out, compile your program, and execute it 3 times with the values: 7 and 5, then 2 and 4, and then 0 and 6. Exit the script mode, print out your script file, and include it in what you hand in for grading.

2.6 Step 4: Batch Mode Using Unix

Sometimes, we do not want to force the user to input all of the data interactively, especially when we have lots of data to be processed. Imagine typing 10,000 data items each time you

run your program! Instead, we would like to type the data in a file one time, a file which we usually refer to as the data file, and then have our C program get the input values from the data file rather than from the user's terminal. Sometimes, we also do not want to print the results to the screen, but would rather output it to a file, which we refer to as the output or result file.

For this lab, perform the following steps to have your `lab2.c` program use file input/output rather than interactive input/output:

1. Create 3 different data files, one for each pair of input values used in the interactive session. To create a data file called `lab2.data1`, use pico to create the file, then type the following into the file:

7 5

That is all that the file contains, just the data numbers, with a space between them! Create similar data files `lab2.data2` and `lab2.data3` for the other pairs of numbers above.

2. Copy your interactive `lab2.c` file to create a new `lab2df.c` file which you will modify. You want to copy, not move, the file, so you can keep a version of the interactive C program.

3. Modify the `lab2df.c` program file to remove the prompts to the user. Do not modify the printing of the results or the inputting of the actual values for num_pears and num_oranges. Now, your C program is ready to obtain input from a file, and is able to print to either the screen or a file.

4. Compile your new program, and execute it with the following line rather than just typing `a.out`:

```
a.out < lab2.data1
```

Your program should be inputting the data from the `lab2.data1` file rather than the screen, but still printing the result lines to the screen. None of the interactive prompts should be printed on the screen.

5. To print the results to a file, execute the program again by typing:

```
a.out < lab2.data1 >! lab2.results1
```

This command says take the input from the file `lab2.data1`, and put the output in the file called `lab2.results1`. The ! insures that if a file named `lab2.results1` already exists, the execution of this command will replace the old contents with the output of this program execution. If you do not type the !, and a file named `lab2.results1` already exists before this command, then the command will not work. Instead, you will get an error message "File already exists". There should be nothing printed to the screen. You should just get

the Unix prompt displayed as if nothing happened. To check to see if the program executed correctly, cat the lab2.results1 file. You should see your output displayed!

6. To demonstrate that you have successfully mastered file input/output, start a script session, cat your modified program, remove the old a.out file, recompile the modified program, execute it 3 times using the three different data files and sending the output to three different files, then cat each output file, and finally exit your script session, and print out your script file to hand in.

2.7 Follow Up

For practice with both vi and working with the C language, do one of the following problems at the end of Chapter 2 in Deitel: 2.17, and 2.19.

2.8 What to Hand In:

Hand in all script from above, stapled as one package, with your name and section number clearly on the front.

Chapter 3

Lab - Expression Evaluation, Variables, and Assignment

3.1 Goals

This lab is an exercise to familiarize you with the following:

- Unix directories
- The use of variables and assignment to store information.
- Arithmetic expressions in C.
- Operator precedence in C.
- Use of the If statement in C (small intro).

3.2 Reference Materials

Appendix A - Useful Unix Commands

Afzal, Chapters 4, 5.

Chapters 2 of Deitel.

3.3 Step 1: Using Directories

Note: Read the section in Afzal on Unix directories before doing this part of the lab.

The files in your unix account (*strauss and copland*) are organized in directories. You can think of directories as *folders* in PC and Mac environments. Some directories are created for you - for example, every user has a *home directory*, and most users have a directory called *mail*. When you login you are in your home directory. You can see what is in a directory by using the `ls` command. For example, in the last lab you entered the command `ls mail` - try it again. The `mail` directory is used by `pine` to store all of your mail. Directories are very useful for keeping related files in one place.

Perform the following sequence of commands to make a new directory for all of your CISC 105 lab files.

1. First, be sure that you are in your home directory by typing the following two commands
   `cd`
   `pwd`

2. Now make a directory called `CIS105Labs` by typing
   `mkdir CIS105Labs`

   (mkdir stands for "make directory")

3. Verify that directory (folder) named `CIS105Labs` is there by typing
   `ls`
   and noting the output.

4. Now move all the lab files into this new directory. Do this with the Unix `mv` command. For example, type

   `mv lab2.c CIS105Labs`
   to move the file `lab2.c`. Repeat this for each file that is related to CISC 105 lab work.

5. Type `ls` again and notice that the lab files are now hidden. Where are they? Type the command `ls CIS105Labs`, and you should see them.

All of your lab files are now in one spot, and not cluttering up your home directory. Now, all you have to do is make sure that all *new* CISC 105 files are created in the directory `CIS105Labs`. To make this happen, simple change to this directory before beginning your work. For example:

```
login

cd CIS105Labs

...do your work ...
```

```
logout
```

Note: the directory you are in is called the *working directory*. At any time you can determine your working directory by type the command `pwd`, and you can go to your home directory by typing the command `cd`. Read the descriptions of the directory-related commands in the Appendix A.

Note: When you download files for this course using Netscape, save them in directory `CIS105Labs`.

3.4 Step 2: Variables and Assignment

1. Copy the file called `lab3a.c` into your `CIS105Labs` directory. Here is the relevant Unix copy command:

```
cp /www/htdocs/CIS/105/Labs/lab3a.c .
```

Note: The dot (.) at the end is important!

You now have a file in your home directory that is a replica of my `lab3a.c` file. Type `ls` to make sure it is there. If it is not, try getting the file again. This file contains the following C program:

```
/*************************************************************************
 * Programmer:                                                           *
 * Course: CISC105                                                       *
 * Section:                                                              *
 * Lab Time:                                                             *
 * File: lab3a.c                                                         *
 * Date:                                                                 *
 * Title: Variables, assignment, and expression evaluation               *
 *                                                                       *
 * This program performs computations, input, output, and assignment.    *
 *                                                                       *
 *************************************************************************
 */

#include <stdio.h>

main() {
   /*declaration of variables */
   int data1, data2 = 0, data3 = 0; /*user's data values  */
   int sum;          /*sum of user's data values       */
   int result;           /*result of complicated expression*/
```

```
/*interactive session to obtain data values for computation*/
printf("Enter first data value: ");
scanf("%d",&data1);
printf("Enter second data value: ");
scanf("%d",&data1);
printf("Enter third data value: ");
scanf("%d",&data1);

/*print out all current values of data variables to check interactive session code*/
printf("Just after interactive session:\n");
printf("Variable values: data1=%d data2=%d data3=%d \n",data1,data2,data3);

/*perform calculations*/
data3 = data1 + data2 + data3;

/*print out all current values of all variables to check computation*/
printf("Just after computation of sum of the data values:\n");
printf("Variable values: data1=%d data2=%d data3=%d sum=%d\n",data1,data2,data3,sum);
}
```

2. Examine the program, and make sure that you understand what it is doing. Compile the program by typing `cc lab3a.c`. Execute the program by typing `a.out`. Use the values 5, 8, and 2 when prompted for the three data values. Observe what the program does now. In particular, observe the values that you input versus the values output for each of the data variables.

3. The programmer actually intended for each new data value to be stored in a different variable, in particular, the three variables data1, data2, and data3. The program currently does NOT do this. Edit the program to achieve this intended action. Recompile the program, and execute it again. Make sure that the printed values for data1, data2, and data3 match the values that you, the user, type as input during the interactive session of execution.

4. Now, notice from observing the printed values during execution that the values printed for one of the variables has been changed after the computation section of the program (i.e., after the section commented as "perform calculations". The programmer did not intend to change any of the variables data1, data2, or data3 throughout any part of the execution of the program. These variables should remain unchanged throughout the entire execution. Edit the current version of the program so that the sum of the data values is still computed, but does not erase the data values already stored in data1, data2, and data3. Recompile the program, execute it again, and observe that your actions corrected the situation.

5. Lastly, add statements to the end of the program to compute the following and print out the results of each computation. These computations should not change the values stored in data1, data2, and data3. That means that you will have to declare more variables to store the results of each of these computations. The printf statements should print out the values of these new variables as well as data1, data2, and data3 to insure that you did not

change their values during the computation. Compile and execute your program with the same input values as before, and insure that it performs as you expect.

a. compute and print twice the value of data1

b. compute and print the value that is 1 more than data1

c. compute and print the square of data3

3.5 Step 3: Expression Evaluation and Operator Precedence

1. Copy the file called `lab3b.c` into `CIS105Labs` home directory. This file contains the following C program:

```
/*************************************************************************
 * Programmer:                                                           *
 * Course: CISC105                                                       *
 * Section:                                                              *
 * Lab Time:                                                             *
 * File: lab3b.c                                                         *
 * Date:                                                                 *
 * Title: Expression evaluation                                          *
 *                                                                       *
 * This program performs expression evaluations and output.              *
 *                                                                       *
 *************************************************************************
 */

#include <stdio.h>

main() {
/*declaration of variables */
int data1, data2, data3;     /*three data values*/

/*initial assignment of 3 data values for computation*/
data1 = 4;
data2 = 3;
data3 = 9;

/*print out original values of data variables */
printf("Original variable values: data1=%d data2=%d data3=%d \n",data1,data2,data3);

/*add code here for lab*/

}
```

2. Add statements to the end of this program to compute and output the product of

the three data values, in addition to printing out the original data variables, which should remain unchanged.

3. Add statements to the end of this program to compute the average of the three data values, and print out the result, in addition to printing out the original data variables again, which should remain unchanged after the computation. This average should be computed as a float.

4. Add statements to the end of this program to compute the sum of the two values: the average of data1 and data2 and the average of data2 and data3. Then, add a statement to print out the result. The original data variables should remain unchanged.

Recompile your final program, execute it, and insure that it works correctly.

3.6 The C If statement

1. Use *vi* to enter the following program into a file called `lab3c.c` - make sure you are in the `CIS105Labs` directory.

```
/*
*****************************************************************
*  Programmer:                                                  *
*  Course:       CISC105                                        *
*  Section:                                                     *
*  File:         lab3c.c                                        *
*  Date:                                                        *
*  Title:        payroll with overtime                          *
*                                                               *
* This program computes weekly pay for an employee, given the   *
* hours worked and the rate of pay.                             *
*                                                               *
*****************************************************************
*/

#include <stdio.h>

main() {
   /*Declaration of variables*/
   int hours;    /* number of hours worked */
   float rate;   /* rate per hour*/
   float pay;    /* total pay */

   /* get values for hours and rate */
   printf("Enter hours worked:");
   scanf("%d", &hours );

   printf("Enter hourly rate:");
   scanf("%f", &rate );
```

```
    /* compute total pay and display answer */
    pay = hours * rate;

    printf("Total pay is %f \n", pay);
}
```

2. Add a variable to hold the overtime part of the pay - call it `overtime` - it should be a float. Declare a second new variable to hold the regular pay.

3. Overtime is computed as follows: if the hours worked is greater than 40, then the rate for the hours over 40 is 1.5 times the regular hourly rate. Add statements to the program so that it properly computes total pay, including overtime. If hours is less than or equal to 40, then overtime should be set to zero. Regular pay is the pay earned on the first 40 hours of work.

To do this, you should use a simple, one alternative *if* statement. If statements are covered in Chapter 4 (which we may not get to for awhile in class)! Fortunately if statements are *easy* to understand. To see the general idea *and examples* of the if statement in C, read Section 4.3 of the text book. This section begins on Page 158.

4. Change the output section of the program so that it produces the following.

```
Hours worked      48
Hourly rate      $ 8.75
Regular pay      $350.00
Overtime pay     $105.00
------------------------
Total pay        $455.00
```

3.7 Practice Problem

1. **Problem Statement:** Variables occupy space in memory - each type of variable may require a different amount of space, and the amount can vary for different computers[1]. Write a small program to determine the size of various data types.

2. **Program Development:** Following is a short outline of the little program. Fill in the missing C statements.

```
/* put a comment block at the beginning*/



/* include the standard header file */
```

[1]The usual *unit of measurement* for space on a computer is the *byte* (standardized at 8 bits at this point in time). Your output will be in terms of bytes.

```
/* open the main() function */

/* declare one variable for each of these types: short, int, long, char,
  float, and double */




/* use the  sizeof function to compute and the  printf
  function to display the memory size for each variable

/* close the  main function */
```

Use *vi* to enter the program into file called `lab3d.c`; then compile and run the program.

3. **Compile and Debug** Compile your program, and use the editor to fix any errors.
4. **Execute:** Make a script file for hand in, in which you cat your source file, and run the program.

3.8 What to Hand In

Before submitting anything, please be sure that you have edited each of the program files lab3a.c,lab3b.c and lab3c.c to contain your name as the programmer. When you are sure your programs work, make three script files, lab3a.scr, lab3b.scr and lab3c.scr, in the following ways:

```
pwd
script lab3a.scr
cat lab3a.c
rm a.out
cc lab3a.c
a.out
```

Input values 5, 8, and 2 when prompted for them during execution here.

```
exit

script lab3b.scr
cat lab3b.c
rm a.out
```

```
cc lab3b.c
a.out
exit


script lab3c.scr
cat lab3c.c
rm a.out
cc lab3c.c
a.out

exit
```

Input 48 for hours and 8.75 for hourly rate.

The first script file will show us your final version of the first program, which is contained in file lab3a.c, show us that it compiles correctly, and executes as desired. The second script file will show us your final version of the second program, which is contained in file lab3b.c, show us that it compiles correctly, and executes as desired. Laser print [2] and submit all four script files as follows.

```
cat lab3*.scr > handin
qpr -q whlps handin
```

[2] Use the printer name for the printer in your lab; here we are illustrating with the printer name for the Willard 009 lab room.

Chapter 4

Lab - Control Structures - I

4.1 Goals

This lab is an exercise to familiarize you with:

- Simple conditionals (if).
- Relational and logical operators.
- Two-way conditionals (if-else).
- While loop.

4.2 Reference Materials

Chapter 3 of Deitel (pp 56-96).

4.3 One-way Conditionals and Logical Operators

Review pages 62-66 (section 3.6) of Deitel before doing this part of the lab.

1. **Problem Statement:** Students in a course take three exams. The instructor would like a program to compute the student's average and letter grade. The letter grade is assigned as follows: 90-100 'A', 80-89 'B', 70-79 'C', 60-69 'D', less than 60 'F'. Your job is to write a program to do this - for one student at a time.
2. **Program Development:** - following is an outline for your program. Fill in the missing C statements to complete the program - then use *vi* to create a source file - called `lab4a.c`.

```
/* put a comment block at the top */


/* include the needed C library header files */


/* open the main function */



/* declare variables to hold the three test scores, and a variable
  to hold the computed average - assume floating point values */



/* prompt and input the 3 test scores */



/* compute the average */



/* write a series of if-statements to determine and print the average
along with the correct letter grade. */



/* close main */
```

3. **Compile and Debug:** Now compile the program, note error messages, and debug your program (using *vi* to make needed changes to the source file. Once the program works correctly, make a script file to hand in - containing a cat of the program source file, a compile and several runs. Call the script file `lab4a.scr`.

4.4 The While loop

Review pages 69-75 (section 3.9) of Deitel before starting this part of the lab.

1. **Problem Statement:** The process of finding the largest number (i.e. the maximum of a group of numbers) is used frequently in computer applications. For example, a program that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write the program, using the following outline.

2. **Program Development:** Following is an outline of the program - fill in the required C statements to complete the program - then enter the program into source file called `lab4b.c`.

```
/*  put a comment block at the beginning of the program */


/* insert the needed header files

/* open the main function

/* declare a variable to hold the current number of units, and
   a variable to hold the maximum number found so far - assume
   integer values.  Start the maximum with a 0. */



/* prompt and read in the first value



/* start a while loop with a stopping condition of value
   less than or equal to 0 */



/* compare current value with maximum value and update
   maximum value */



/* prompt and read in next value */

/* close the loop */
```

```
/* output the maximum value, and close the main function */
```

3. **Compile and Debug:** Once you have created the source file, compile, note error messages - and use *vi* to correct the source file. Do this until the program compiles without errors.

4. **Execute:** Run the program, and verify that it is working - if there are run-time errors, fix the problem and recompile and run again.

5. **Script:** Once the program runs correctly, make a script file called `lab4b.scr`, containing a cat of the source file, and several runs of the program.

4.5 Another While Loop Problem

1. **Problem Statement:** The process of finding the sum or average of a list of numbers is also used frequently in computer applications. For example, a program that determines the class average for an exam, might input the exam scores one at a time, counting and summing as it goes. Then it could compute and print the average or the sum.

2. **Program Development:** Following is an outline of the program - fill in the required C statements to complete the program - then enter the program into source file called `lab4c.c`.

```
/*  put a comment block at the beginning of the program */


/* insert the needed header files

/* open the main function

/* declare a variable to count the number of exams, and
   a variable to sum the exams - and set both of these to 0.
   Declare a variable to hold the computed average, and
   a variable to hold one exam value.
   (Use good variable names!)  */




/* prompt and read in the first exam value
```

```
/* start a while loop with a stopping condition of value
   less than 0 */



/* Count and sum the current exam value */



/* prompt and read in next value */

/* close the loop */

/* compute the average, and display the results */

/* close the main function */
```

3. **Compile and Debug:** Once you have created the source file, compile, note error messages - and use *vi* to correct the source file. Do this until the program compiles without errors.
4. **Execute:** Run the program, and verify that it is working - if there are run-time errors, fix the problem and recompile and run again.
5. **Script:** Once the program runs correctly, make a script file called `lab4c.scr`, containing a cat of the source file, and several runs of the program.

4.6 Follow Up

For practice with vi and with C program statements, do one of the following problems at the end of chapter 3 in Deitel: problems 13, 25, or 36. Make script files as usual.

4.7 What to Hand In

Submit all of your script files - stapled together as one package-with your name and section number clearly indicated on the front page. (Due at the beginning of your next lab.)

Chapter 5

Lab - Control Structures - II

5.1 Goals

This lab is an exercise to familiarize you with:

- Single, counter-controlled `while` loops.
- Single `for` loops.
- Program Development with loops.

5.2 Reference Materials

Chapters 3 and 4 of Deitel.

5.3 Counter-Controlled Loops with the `while` Statement

1. For a warm up problem, let us construct a program with a counter-controlled while loop, and look at the loop counter values. Use the following outline as a starting point.

```
/* put the usual comment block here */



/* add necessary header files */
```

```
/* open the main function */


/* declare a variable to be used as a counter, and give
   it an initial value of 0.  */


/* Construct a while statement with a to do a loop
   as long as the counter is less than 12.  */
   /* Inside the loop, do the following: */

        /* print the value of the counter
           one value per line of output */

        /* Increment the counter by 1 */



/*  After the loop is completed, print out
    the current value of the counter.  */


/* end main function */
```

Store the program in file called `lab5a.c`, compile and run, and make a script file as usual - called `lab5a.scr`.

2. Make a copy of the first program - copy into file `lab5b.c`. Edit the file and change to ending value in the while statement to 28, and the increment value in the loop body to 5. Compile and run and make a script file called `lab5b.scr`.)

5.4 Counter-Controlled Loops with the `for` Statement

1. **Problem Statement:** The local weather station records temperature data on each hour. Write a programs that will read 24 temperature values for a given day, and compute and print the high temperature, the low temperature and the average temperature for that day.

2. **Program Development:** Following is an outline for the program; fill in the missing C statements to complete the program; enter the program into source file called `lab5c.c`.

```
/* put the usual comment block here */



/* add necessary header files */


/* open the main function */


/* declare variables for temperature, low temperature, high
   temperature, sum, and average.  Also declare a variable
   to count from 1 to 24.  Initialize variables.  */

/* print an introductory message - with instructions
   for the user. */

/*  Set up a for loop to do 24 iterations */
      /* read value */
      /* add to the sum */
      /* update maximum */
      /* update minimum */
/* end loop */

/* compute the average */


/* print results */


/* end main function */
```

3. **Compile and Debug:** - Once the source file is created, compile and debug. When the program compiles without errors, test it to verify that it runs correctly - using your own data.

4. **Data:** (Note: to make things easier, study section 7.2 (pp. 165-167) of Afzul - on Unix redirection.) Use *vi* to create a file called `temps.data`, and enter 24 temperature values. Note that data files are **not** source files - no comments, just data. The data must be in order, but may be any number of values per line in the file.

5. **Execute:** Now run the program, using your data file - and verify the correctness of the results. Then, make a script file called `lab5c.scr` in which you cat the program, cat the data file, and run the program using the data file.

5.5 Another Counter Controlled Loop

1. **Problem Statement:** Consider the weather program written in the preceding section: sometimes the number of data values is not 24 - there may be some missing data values, or there may have been extra readings taken on a given day. (Some days there might be 20 values, other days 24, and some with 27, for example.)

2. **Program Development:** First, copy the source file `lab5c.c` into file `lab5d.c`. Modify the program so that it first reads in the number of values - into variable called `Nvalues`. Then change the loop so that it counts `Nvalues`, instead of 24; and change the calculation for the average.

3. **Data:** Make three data files - called `t1.data`, `t2.data`, and `t3.data` - start each one as a copy of the original `temps.data`. Make one file have 20 values, one have 24 values, and the other have 27 values, and put the number of values (20, 24, or 27) at the beginning of the file.

4. **Execute:** Now run the program on each data file and verify that it works correctly. Then make a script file, called `lab5d.scr` in which you cat your program, and the three data files, and make the three program runs.

5.6 Follow Up

For additional practice creating your *own* programsa, do problem 26 at the end of chapter 4 in Deitel - make a complete running program and the usual script file for turn in - with your written answers to the follow-up questions.

5.7 What To Hand In:

All script file created above - including the homework - stapled as one package - with your name and section number clearly indicated on the front page.

Chapter 6

Lab - Practice Program

6.1 Goals

The goal of this lab is to give you practice developing and writing a complete C program:

- Develop an algorithm and plan a program based on the algorithm
- Write a C program based on the algorithm
- Identify and correct syntax errors
- Test a program to see if it contains logic errors.

After completing this lab, you will be ready to begin programming project #1.

6.2 Reference Materials

Before beginning this lab be sure you have studied chapters 1-4 of Deitel and that you have completed all of the preceding labs.

6.3 Problem Description

The **I & S Cube Tax Co.** has just hired you to write the software for their tax computation schedule. They want the program to perform the following actions.

1. Print an informative, but short, wake-up greeting to the tax consultant as they begin to execute the program.

2. Ask the tax consultant if they are ready for their first client, and wait for the tax consultant to type 1 for yes or 0 for no. If they type 0, end the execution; otherwise, begin to input data about the first client's taxes (described below).

   After the first client has been processed, then ask the tax consultant whether they want to process another client. If they type 0, end the execution. If they type 1, repeat the processing for this new client. The processing of clients should continue to repeat until the tax consultant types 0 in response to being asked whether they want to process a new client. Therefore, the program should handle any number of clients, maybe 3 on one execution, and 100 on another execution.

3. For each client, interactively obtain the following information from the tax consultant, by a nice, user-friendly interactive session:
   (1) the total salary for 2001 for this client
   (2) the number of dependents for this client
   (3) the total amount of deductions for this client
   (4) the amount of income tax withheld in 2001 for this client

4. After this information is all input by the tax consultant in response to your program prompting them for it, then compute the client's total tax due, or the amount of the refund.

   Some facts that you need to know to compute the tax are the following:

   - The allowance per dependent is $2,500.
   - The taxable income is computed by subtracting the total deductions and the total dependent allowance from the salary.
   - Note that if the taxable income is negative, it should be set to 0.
   - The tax is computed using the following tax table. [1]

Taxable Income	Tax
less than 4000.	0.00
4000. to 9000.	0.00 plus 1% of income over 4000.
9000. to 18000.	50.00 plus 2% of income over 9000.
over 18000.	230.00 plus 3% of income over 18000.

5. Print out a record for the client to the screen. A client's record should look like the following format:

   ****

[1] Since the tax function is a continuous function, it matters not which bracket you include 9000 in, for example.

```
I & S Cube Tax Co.
total salary:          $32500.00
number of dependents:      3
total dependent
      allowance:        $7500.00
amount of deductions: $10125.00
                      ----------
taxable income:        $14875.00

taxes from tax table:    $167.50
total tax withheld:      $220.30

amount of refund:         $52.80


*****************************************
```

6. After all clients for today have been processed, print a friendly message to end the day for the tax consultant.

6.4 Program Development

Follow the general strategy that we have used in previous labs for developing programs. Here is a partial outline to get you started.

```
/* the beginning comment block.....*/



/* and header files */

/* begin main */

/* make some variables */


/* display a startup message */

/* start while loop - tax consultant loop */

           /*prompt for client info*
```

```
            /* compute tax info, and
               display results */

/* end while */

/* end program */
```

When you are ready to type in your program, enter your code into file `taxes.c`.

6.5 New and Improved Version

Copy your program and extend it in two ways.

(1) Extend the program so that it handles erroneous input nicely. That is, whenever the tax consultant inputs an invalid number for any of the questions, your program should tell the user that the input is wrong and repetitively prompt for a legal value until they input a legal value. Hint: This will involve adding a WHILE or DO-WHILE loop around the code that int eractively inputs a value.

(2) Extend the program to display a total number of clients processed and the total tax collected for that day. By *total tax*, we mean **both** the tax due and the taxes withheld. Have your program print these totals *after* the last client record is printed.

Note: You can assume that the tax consultant inputs the correct type for each question , that is, numbers, not letters. You should check to be sure that the input is not negative numbers for the salary, deductions, and tax withheld, and that the number of dependents is at least 1. You should also make sure that the tax consultant inputs either 0 or 1 in respon se to whether they want to process another client.

6.6 What To Hand In

Note: Remember to follow the programming guidelines listed in Part III.

Run your program several times and satisfy yourself that it produces correct answers. Be sure to try typical values for the answers to the program's questions, as well as legal, but, what we call boundary values, such as 0 and 1 for some of the answers. For example, try 1 for number of dependents, and try a combination that produces a negative taxable income (the program should use 0 in this case!).

When you are sure your program works, make a script file by doing the following:

```
script taxes.scr
cat taxes.c
cc taxes.c
a.out
exit
```

Show executions with the following input cases:
(1) only one client
(2) no clients
(3) 3 clients
with the following purchases:

	client value	client value	client value
salary:	40500	12400	22100
dependents:	4	1	3
deductions:	8950	2211	3505
tax withheld:	550	125	53

Hand in a copy of your script file printed on a laser printer.

Chapter 7

Lab - User-Defined Functions

7.1 Goals

This lab is an exercise to familiarize you with:

- How to make and use functions.
- Function Definitions, Prototypes, and Calls.
- Passing information to functions via parameters and global variables.
- Program Development with Functions.

7.2 Reference Materials

Chapter 5 of Deitel.

Afzal, Chapter 6.

7.3 Basic Function Concepts

1. There are two kinds of functions: those which return a value (of types like float, int, double, ...) and those which do not return a value (type void). IF you need a value (e.g., a numerical value) RETURNED from a function (lets name it fun1 to be able to show examples):

   (a) It must have a suitable type (other than void), e.g.,

   ```
   float fun1(int n, float x){...};
   ```

AND

(b) You will NOT get any use of its returned value if your call on it (say in main or in some other function) looks like:

```
fun1(6, 42.0);
```

INSTEAD you have to assign its value to some variable, e.g., as in:

```
y = fun1(6, 42.0);
```

Of course, then, y has to have been typed appropriately.

2. If you have some values of variables assigned in one function, and you need to USE those values in some OTHER function, then you need to supply those variables as ARGUMENTS in a call of the SECOND function. Here's an example. Suppose in main we have:

```
int n;
float x,y;
```

Suppose main assigns values to n, x, and y AND that fun2 needs to know these values (and no other values) to do its thing. Lets suppose for THIS example, fun2 returns nothing. Then its typing could look like:

```
void fun2(int m, float u, float v){...};
```

AND, when main calls fun2 the call (in main) would look like:

```
fun2(n, x, y);
```

It should NOT look like:

```
fun2();
```

since, then, the values fun2 needs to do its thing are not made available to it (and you'd get an error message about mismatch in number of arguments). (To get rid of the argument mismatch error, you could type fun2:

```
void fun2(void){...};
```

but, then, fun2 does NOT get the information about the values of n,x,y it needs, SO give fun2 3 arguments - each place instead of 0 arguments each place.)

The function call should also NOT look like:

```
fun2( int n, float x, float y );
```

since the typing of arguments is done in the function prototype and function header; NOT in the function CALL.

3. Be sure to type any local variables you are using INSIDE a function; the typing for them must be done, then, INSIDE that function to which they are local. Here's an example.

   Suppose you want a function definition like:

```
float fun3(int k, float z){
   answer = k * z;
return answer;}
```

   THIS will not work; however the following will work.

```
float fun3(int k, float z){
   float answer;
   answer = k * z;
return answer;}
```

   provided you handle the prototyping, etc.

   Now, with these important background points covered, let's try some lab exercises.

7.4 Function Definitions, Prototypes, and Calls

1. **Problem Statement:** We wish to process purchase information for a retail music store. The items are CD's with various price levels: A - 12.99, B - 13.99, C - 14.99, S - 9.99, D - 16.99. The customer will enter the code and the number of items purchased; the program will compute the total price.

2. **Program Development:** We'll use two functions to orgranize the program solution:

   - A void function with no arguments - called `showInstructions` - which displays a helpful message to the user - explaining how to use this program.
   - A float function with two arguments (a char code for the cd price level, and an int for the number of cd's purchased) - called `computePrice` - which computes and returns the total price.

   Here is a basic outline for the program; fill in the missing C statements to complete the program. Then enter your code into source file called `lab7a.c`.

```
/* put the usual comment block here */

/* add includes for the necessary header files */

/* put function prototypes here: one for function
   called showInstructions, another for function
   called computePrice. */

/* open the main function */

/* declare variables for number of cd's, the cd code,
   and total price */



/* Use your showInstructions function to display
   instructions to the user */



/* prompt and read in the number, and the code *



/* compute the total price, using your computePrice function *


/* Display results - the number, the code, and the total price */

/* close the main function */

/* user-defined function definitions usually
  follow the main() definition */
/* open the showInstructions functions */

/* write the statements for this function *







/* close the showInstructions function */
```

```
/* open the computePrice function */

/* use a switch statement to compute and return
   the total price, based on the cd code */









/* close the computePrice function */
```

3. **Compile and Debug:** Compile your program and note any error messages - debug until the program compiles without errors.

4. **Execute:** Once your program compiles without errors, run the program several times to verify that it runs correctly. Then make a script file called `lab7a.scr` in which you cat the program, and show several runs.

7.5 A Practice Program

1. **Problem Statement:** A *prime number* is a positive integer that is divisible only by 1 and itself. For example, 2, 3, 5, 7 are prime numbers, but 4, 6, 8, 9 are not. Write a program to find prime numbers [1].

2. Program Development: It is convenient to have a function that can test a single number for prime-ness, called `isPrime`, with one int argument, and returning 1 if prime, and 0 if not prime. First write this function and test it with a small main (sometimes called a "driver" program). Here is an outline to start you off.

```
/* the usual comment block goes here.... */

/* and header files */


/* then function prototypes */

   int isPrime(int);
```

[1] This exercise is based on problem 27 in Chapter 5 of Deitel

```
/* then open main */

/* make a variable to hold an integer;
   prompt and read in a value */


/* us isPrime to test for prime-ness
   and output an appropriate message */

/* close main */

/* definition of isPrime goes here: */

   int isPrime( int number ) {









}
/* end of program */
```

Enter your program into source file called `lab7b.c`.

3. **Compile and Debug:** Compile and debug until the program compiles without errors. Then make a script file called `lab7b.scr` in which you cat your program and run it several times.

4. Modify: Copy file `lab7b.c` into file `lab7c.c`, and make the following modifications.

   (a) Change `main` so that it checks a range of integers form `nStart` to `nEnd` (eg. 300 to 500), and prints out just those integers that are prime. Have it prompt the use for the values of `nStart` and `nEnd`.

   (b) Change `isPrime` so that it check divisors from 2 up to the square root of the number[2].

5. Compile and Debug: Once these changes have been made, compile and debug your new program, until it compiles without errors.

6. **Execute:** Verify that your new program works - by giving it a range of small numbers that you can verify by hand. Once you have a working program, make a script file called `lab7c.scr` in which you cat your program and run it 3 times with the following ranges.

[2] Can you prove that this is sufficient to determine prime-ness?

(a) 50 to 100

(b) 10000 to 10100

(c) 2000000000 to 2000000100

7.6 Follow Up

For additional practice, do problem 15 at the end of chapter 5 in Deitel. [3].

7.7 What to Hand In

Hand in all of the script files from above - including your homework, stapled as a single package, with your name and section number clearly indicated on the front page.

[3]When using functions from the math library, don't forget to include the header file `math.h`, **and**, don't forget to compile using the `-lm` switch.

Chapter 8

Lab - Debugging

8.1 Goals

This lab is an exercise to familiarize you with:

- Helpful Hints on Debugging Programs.

8.2 Reference Materials

Chapter 13 of Deitel.

The `lint` manpage.

8.3 Errors you may encounter in a C program

While writing C programs, you will encounter many kinds of errors. The first and the easiest to find and fix are syntax errors found at *compile* time. Almost all of you must have seen these errors. Then there are *semantic run-time* and *logical* errors which are not as easy to find and fix. These are the errors that occur after your program compiles. These result in incorrect output. In this lab we shall discuss *debugging* a program. The term *bug* is based on thinking that bugs or insects have gotten into your program, and that's the reason for the error.[1] The very first computers were so huge that one time an error was actually caused by a moth that flew inside and appropriately the term *debugging* refers to taking the bugs out of the computer. In the context of a *bug* in your program, *debugging* would mean removing the *logical* error that is causing the program to run incorrectly.

[1] For a little note on the first "bug", see `http://www.cs.yale.edu/homes/tap/Files/hopper-wit.html`

8.4 Debugging By Tracing Execution

The easiest way to debug a program is to place extra `printf` statements in your program so that you can trace the execution of your program and watch how the value of each variable changes with the execution of each statement in your program. There is also a way to run your program step by step using a debugger (eg., `dbx`), however for this lab we shall only use `printf` statements to debug a program.

1. Copy the file `debug1.c` into your own home directory. This file contains the following C program:

```
/*************************************************************************
 * Programmer:                                                           *
 * Course: CISC105                                                       *
 * Section:                                                              *
 * Lab Time:                                                             *
 * File: debug1.c                                                        *
 * Date:                                                                 *
 * Title: Debugging                                                      *
 *                                                                       *
 *************************************************************************
 */

#include <stdio.h>

/* This program focuses on debugging common errors via tracing.         *
 *    Sum the even integers < n, i.e., output the sum 0+2+4+...+m       *
 *    where m is the largest even number < n, n is input by the user    *
*/

main() {
   /*declaration of variables*/
   int number;        /*user-entered value of n*/
   int i = 0;
   int sum = 0;       /* sum of even integers >0 and < n */

   printf("Enter a number: ");
   scanf("%d", number); /* read the number n */
   while ( i < number ); {
      sum = sum + i;
   }
   printf ("The sum of the even numbers < n is ", sum);
}
```

This program contains several common programming errors. DO NOT CORRECT ANY ERRORS that you may notice. Now compile the program. Note that this program compiles fine. If you run this program, you will notice that this program stops with an error. Here is an example run of this program:

```
Enter a number: 5
```

There may be a segmentation fault error message - a message from the operating system (UNIX), indicating that your program has tried to access some part of the computer memory that it is not allowed to access. Now, we want to determine which is the last statement executed when the program had a segmentation fault. Normally, we want to find out exactly on which statement the program stops. So, if we put `printf` statements throughout the program, we should be able to find out which line is causing the problem. The printf statements should indicate the following of information: (1) How far did execution get? That is, only the printf statements that are actually executed before the error occurs will be printed. Others will not be printed. (2) What were the values of the important variables as the program executed?

Here is one way to put in these debugging printf statements:

```
#include <stdio.h>
#define DEBUG 1

main()
{
/*declaration of variables to be used in this computation*/

   int number;
   int i = 0;
   int sum = 0;      /* sum will store the sum of even integers >0 and < n */

   if (DEBUG) printf("DEBUG.1: Execution reached just before interactive input session. \n");
   printf("Enter a number: ");
   scanf("%d", number); /* read the number in */

   if (DEBUG) printf("DEBUG.2: Just after interactive session, sum = %d \n", sum);

   while ( i < number );
   {

      if (DEBUG) printf("DEBUG.3: Just beginning loop body, sum = %d  i=%d\n", sum, i);
      sum = sum + i;
      if (DEBUG) printf("DEBUG.4: After sum computation, sum = %d  i=%d\n", sum, i);
   }
   if (DEBUG) printf("DEBUG.5: Just after loop exit,  sum = %d\n", sum);
   printf ("The sum of the even numbers < n is ", sum);
}
```

IMPORTANT: Every `printf` used for debugging must end with `\n`.

2. Copy the original erroneous program into a new file called debug-new.c, and modify the new file by typing in the 5 DEBUG statements and the statement `#define DEBUG 1` in the

indicated places as above. Compile and execute the modified program. If done correctly, your program should output:

```
DEBUG.1: Execution reached just before interactive input session.
Enter a number: 5
Segmentation fault
```

Since `DEBUG.2` never got printed, we know the error occurs after printing `DEBUG.1` and before `DEBUG.2`. The error must therefore be either in the `printf` or the `scanf` statement of the interactive session. Create a script file that cat's this version of your program before doing any bug fixes, removes a.out, compiles and executes this version. Print out this script file for grading.

3. Determine the error causing the sementation fault, and fix ONLY this error. Now compile and run the program again. You will notice that your program seems to be "hanging" just after the DEBUG.2 statement is printed. Your program is in a state of infinite execution. The only way to stop it is to hit the Control key SIMULTANEOUSLY with the "c" key. This is called hitting Control C, sometimes written in shorthand as $\hat{C}$. This should bring back the Unix prompt. The `DEBUG.3` statement never got printed. This means that the while loop is being executed indefinitely, but it appears that the statements which look like they are inside the loop are never being executed. This is called an infinite loop.

4. Look carefully at the `while` loop condition line, and fix the error causing this symptom. Compile and execute this version of your program. Your program is still in an infinite loop, but now the loop body statements are clearly being executed over and over again, as the DEBUG.3 and DEBUG.4 statements are being printed continuously. Stop the infinite execution again by hitting Control C. Since DEBUG.5 never gets executed, we know that the error occurs in the loop body.

5. Look carefully at the values of `sum` and `i` in the DEBUG.3 and DEBUG.4 statements that were printed. Fix the one error that causes the loop to go indefinitely, and compile and run again. To test your program, use a small input value so that you can compare the output of the program with your own hand calculations. You will notice that `i` is getting incremented properly inside the loop, but the value of `sum` is clearly wrong. This is a logical error. Does the value of `sum` go wrong inside the loop, or before the loop?

6. Look closely at `DEBUG.2`, `DEBUG.3`, and `DEBUG.4` output. First, decide if the value of sum is okay before the execution of the loop. Then see if only the EVEN numbers between 1 and `n`, not including `n`, are being added. Fix the error in the body of the loop to obtain the correct value of `sum` in the `DEBUG.5` statement after the loop exit. Compile this new version of your program and execute to insure that the value for `sum` at `DEBUG.5` is correct.

7. Finally, you will notice that the original (non-debugging) `printf` after the loop body, which prints out the value of sum, is incorrect. Using our `DEBUG` statements, we know this problem occurs after `DEBUG.5` Find this last error, and fix it. Compile this final version of the program, and execute it with the values 5, 8, and 0. Check your answers against hand

calculations. Create a script file that cat's this version of your program, removes a.out, compiles it, and executes it with these three values as input. Print out this script file for grading.

8. Now, *turn off* the debugging by editing one line of your program file, changing `#define DEBUG 1` to `#define DEBUG 0` Recompile and run this version to get a clean output for your program execution (i.e., without any debugging print's). Create a script file that compiles and executes this version for the input 8 only. Print out this script file for grading. It need not include a cat of your program.

8.5 Using `lint` to find logical errors

There is another program available called `lint` which helps you find errors in your program. It attempts to detect the features in your C program that are *likely* to be bugs. These are potential logic errors; they are not syntactic errors that the compiler will find. For example, use `lint` on the original program that we gave you (by typing `lint debug.c`), you will get the following warnings (note: depending on which machine you execute `lint`, you may see some variations of these messages):

```
debug.c(27): warning: number may be used before set
debug.c(33): warning: main() returns random value to invocation environment
printf returns value which is always ignored
scanf returns value which is always ignored
```

You may not fully understand the messages; that's OK. Most importantly, look at the lines where warnings are issued; in this case, use the editor to look at lines 27, 33 because they are the lines indicated by lint in the message above as possible problems. Assume something is wrong with these lines and try to determine what is wrong. If you do, great. If not, simply go on for now.

From now on in CISC105, you should ALWAYS run `lint` on your programs BEFORE you execute them to help you find possible logic errors.

8.6 Tracing Programs with Function Definitions

1. Copy the file `debug2.c` into your own home directory. This file contains the following C program:

```
/*****************************************************************
 * Programmer:                                                   *
 * Course: CISC105                                               *
```

```
 * Section:                                                              *
 * Lab Time:                                                             *
 * File: debug2.c                                                        *
 * Date:                                                                 *
 * Title: Debugging                                                      *
 *                                                                       *
 * This program focuses on debugging functions via tracing.              *
 *                                                                       *
 *************************************************************************
 */

#include <stdio.h>

/* This program computes and prints the square of the numbers 1 through*
 * 10, inclusive.  It prints the number and the square of the number on*
 * a line, then goes to the next number.  The function square_it is    *
 * called to compute the square as a number, so it is called 10 times.*
*/

main() {
   /*declaration of variables*/
   int number;       /*represents squared value*/
   int i = 0;        /*loop index variable*/

   for (i=1; i<=10;i++) {
       number = square();
       printf(" %d  %d\n",i,number);
   }
}

/*function definition for function square which squares an incoming integer
  and returns this value*/
square_it(int arg); {
   int result;
   result = data * 2;
   return result;
}
```

2. Compile this program. It contains several compiler errors and warnings. Examine the lines that are indicated in the compiler errors. Warnings of the form, `debug2.c:33: warning: data definition has no type or storage class` usually indicate that some name that you are using in the program has not been declared before you are using it in this line.

3. Check that all variables that appear on these lines have been declared, and all function names on these lines have been declared by a function prototype. Fix ONLY these errors at this time, and try to recompile your program. Be careful that you determine whether a variable is a variable name or parameter name, and make the corrections appropriately. You are probably still having almost the same errors, in fact, maybe even more errors. It appears that the name "data" is not being seen as being declared. It is meant to be a formal

parameter of the function `square_it`. There is also a message of the form `syntax error before {`. This should indicate that there is an error in the first line, i.e., the function header, of the function definition.

4. Examine the function header line, and fix the error. Recompile again. This time, there should be no compiler errors. But, now when you compile, you should be getting an error that looks like:

```
ld: Undefined symbol
   _square
collect2: ld returned 2 exit status
```

This means that you have used the name `square` as a function call somewhere in your program, and there is no function in your program file or the C standard library with this name, so the loader could not figure out what function is being called. Fix this error in your program, and recompile again.

5. Now, you should be getting the error:

```
debug2.c: In function ‘main’:
debug2.c:25: too few arguments to function ‘square_it’
```

This indicates that the number of parameters passed to the function `square_it` in a call to this function does not match the number of parameters declared for `square_it` in the prototype and function definition. Fix this problem in the call to `square_it.` Your program should compile with no errors.

6. Now try executing your program. You will notice that it does not print the correct answers for the square of each number. Insert 2 DEBUG statements into the function definition for `square_it`. Also, insert the `#define DEBUG 1` statement in the appropriate place at the top of your program to turn on the debugging feature. DEBUG.1 should go just after the local declarations of `square_it`, and DEBUG.2 should go just before the `return` statement. Both debugging printf's should print out the value of `data` and `result` to check what is happening inside `square_it`. DEBUG.1 tells you that the correct number is passed in as a parameter. DEBUG.2 tells you that the correct result is NOT being calculated. Fix this logical error inside `square_it`. Insert a debugging statement called DEBUG.3 in the main program, just after the call to `square_it` which prints out the value returned from `square_it`, namely the variable `number`. Recompile this program and execute it. When you are satisfied that you are getting the correct values, change the #define statement to TURN OFF debugging, leaving your debugging statements in the program, create a script file that cat's this version of your program with the DEBUG printf's in it, removes a.out, compiles your program, and executes it. Print out this script file for grading.

8.7 What to Hand In

Script files as created above.

Note: You are now ready to begin work on Project #2.

Chapter 9

Lab - Arrays - I

9.1 Goals

This lab is an exercise to familiarize you with the following:

- array processing.
- Handling end-of-file.

9.2 Reference Materials

Chapter 6 of Deitel (pp. 196-248).

Page 421 of Deitel.

Afzal, Chapter 7.

9.3 Simple Array Processing

Be sure to read Chapter 6 of Deitel before you begin. Remember that arrays contain all the same kind of data (all ints, all floats, chars, etc.). Arrays are a fixed size - and the size must be a constant. The first array position is 0 (not 1); and the last array position is one less than the size. For example, if `A` is an array of size `10`, then

```
A[0], A[1], A[2], ..., A[9]
```

are the elements of `A`.

1. **Problem Statement:** Write a simple program to illustrate declaration and initialization of arrays, and the use of for loops to output array elements.

2. **Program Development:** Following is an outline for a C program - fill in the missing C statements to complete the program. Enter your code into source file called `array1.c`.

```
/* put the usual comment block here */


/* include necessary header files */


/* open the main function */

/* Use one statement to declare a character array
   of size 5 - called Vowels - and initialize the elements
   to 'a', 'e', 'i', 'o', 'u' */



/* Use one statement to declare an integer array called
   primes with size 10, and initialize the elements to
   2, 3, 5, 7, 11, 13, 17, 23, 29, 31. */




/* Use one statement to declare an array of 8 floats called
   Rates with initial vales of 1.25, 3.5, 3.92, 4.75, 5.3,
   6.25, 7.5, 8.15 */



/* Write 3 separate for loops to output the values exactly as follows:
         /* one for loop here: */
         Vowels = a, e, i, o, u.

         /*another for loop here: */
         Primes:
            primes[0]  =  2
            primes[1]  =  3
            primes[2]  =  5
            primes[3]  =  7
            primes[4]  = 11
```

```
            primes[5]  = 13
            primes[6]  = 17
            primes[7]  = 23
            primes[8]  = 29
            primes[9]  = 31

         /* and another one here: */
         Rates = 1.25 3.50 3.92 4.75 5.30 6.25 7.50 8.15

*/
/* close the main function */
```

3. **Compile and Debug:** Compile and debug your program until it works correctly.

4. **Execute:** Make a script file called `array1.scr` in which you cat your program, and run it.

9.4 An Array Application

1. **Problem Statement:** Another common application in software is to process a list of measurements. A program is needed that can process up to 200 measurements (no more than 100 positive values, no more than 100 negative values). The program needs to store to positive measurements in one array, and the negative measurements in another array; and then output the two arrays in side-by-side columns[1]. Note: you may use only the **two** arrays mention - no others! The output could look like this:

```
2.5   -3.8
1.2   -2.6
3.4   -4.2
3.5   -1.1
```

   or this:

```
2.5   -3.8
1.2   -2.6
3.4
3.5
```

   or this:

[1] If you are having difficulty with the output, you *may* simplify it by printing the arrays in sequence (one after the other) rather than side-by-side. To receive *full* credit, you must eventually solve the side-by-side problem.

```
 2.5   -3.8
       -2.6
       -4.2
       -1.1
```

In fact, one of the arrays might even be empty!

2. **Program Development:** Following is an outline for part of the program solution. Complete the program to solve the problem. Then enter your code into source file called `array2.c`.

```
/* the usual comment block ... */



/* and header files ... */

/* define a name for the constant 100 - call it MaxSize

/* open the main function */

/* declare two float arrays of size MaxSize - named PosArray
   and NegArray and declare two counters: nPos and nNeg with
   initial value of 0  and declare a single float to hold an
   input value */


/* set up a loop to read a single float value until end of
   file is reached - use one of the following techniques:
                     while( !feof(stdin) )
             or
                     while( scanf("%f", &value) != EOF )
*/

         /* if the value is positive, store it in the PosArray */


         /* if the value is negative, store it in the NegArray */

/* close the end-of-file loop *

/* write loops to output the arrays */
```

```
/* close the main function */
```

3. **Compile and Debug:** Compile and debug your program until it compiles without errors.
4. **Test Data:** We will need to create several data files in order to thoroughly test the program - they don't have to be large - maybe 20 values or so. Create five test data files, each with exactly one of the following properties:
   (a) the number of positive and negative values is the same
   (b) there are no negative values
   (c) there are no positive values
   (d) there are fewer negative values than positive ones
   (e) there are fewer positive values than negative ones
5. **Execute:** Run your prgram on each test data file; fix any problems; verify that your program runs correctly for all of the test cases. Then make a script file in which you cat your program and the five test data files, and show five runs of your program - on each data file. Call the script file `array2.scr`.

9.5 Follow Up

For additional practice with arrays, and creating your *own* programs, do problem 15 at the end of chapter 6 of Deitel.

9.6 What to Hand In

Hand in all script files created above, including the homework, stapled together as one package, with your name and section number clearly indicated on the front page.

Chapter 10

Lab - Arrays - II

10.1 Goals

This lab is an exercise to familiarize you with:

- More array processing.
- 2-dimensional array processing.
- Passing arrays as parameters to functions.

10.2 Reference Materials

Chapter 6 of Deitel.

10.3 A 2-Dimensional Array Example

1. **Problem Statement:** A magic square is an integer array (2-dimensional) with the same number of rows and columns (i.e., square), with each cell containing a positive integer, and with the property that the sum of each row, each column, and each diagonal is the same number. [1] Here is an example of a 3 by 3 magic square:

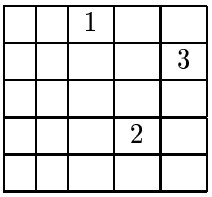
6	1	8
7	5	3
2	9	4

[1] This number is called the magic number. For a 5 by 5 square, using numbers 1, 2, 3, ..., 25, the magic number is 65. (Can you prove this?) What is the *magic number* for a 4 by 4 square filled with 1, 2, 3, ..., 16?

2. Algorithm: An algorithm to generate some magic squares is available - it assumes the size of the row (or column) is odd. To keep things concrete, we'll assume a 5 by 5 square, and fill the cells with numbers 1, 2, 3, ... , 25. The algorithm goes like this:

   - start at any cell and put a 1 there
   - move to the new cell - by moving to the right one column, and up two rows - if you move past an edge, just wrap around to the other side.
   - if the new cell is not occupied, place the next number there; otherwise, go back to the previous cell, and drop down to the cell under that one, and place the next number there.
   - repeat steps 2 and 3 until all cells are filled.

   For example, here is a 5 by 5 with the first 3 cells marked - follow the algorithm to complete the magic square.



3. **Program Development:** Develop a program based on the algorithm; remember to declare an array with 5 rows and columns, declare variables to hold the starting position, and prompt and read in the starting position from the user; make a loop to count from 1 to 25; within the loop mark the cell with the counter value, and move to the new cell - based on the algorithm. Once the square is complete, print it so that it looks nice.

   Store your program code in source file called `magic.c`.

4. **Execute:** Compile and debug `magic.c`. Once it compiles without errors, run it and verify that it runs correctly - fix any problems. Then make a script file called `magic.scr` in which you cat your program, and run it 3 times - with a different starting position each time.

10.4 Functions and Arrays

Remember the following about arrays as arguments to functions. If a function is defined to process an array (say an int array), then the header must contain something like the following:

```
void aFunction( int x[], int Nx )
```

Then, to use the function, your call would look like:

```
aFunction( myArray,  nData )
```

1. **Problem Statement:** A list of integers that reads the same forwards and backwards is called a palindrome. For example, the list 13, 32, 44, 32, 13 is a palindrome; but the list 2, 4, 6, 8 is not. We need a program to check for "palindrome-ness".

2. **Program Development:** We'll write some functions to organize the program solution. Here is a sketch of what you are to do. Fill in the missing statements, and create a source file called `palindrome.c`.

```
/* the usual comment block ... */


/* and header files */


/* prototype TWO functions: Reverse - which takes an integer
                            array of size N, and copies it in
                            reverse order into a second integer array

                            Compare - which compares two integer
                            arrays of size N, and returns 1 if they
                             are exactly the same, and 0 if they are not.
*/

/* open the main function */

/* declare two integer arrays */

/* prompt and read in a list of integers
   from the user */

/* Use Reverse to put the reversed array into your
   second array */

/* Use Compare to determine if the two arrays are
   the same - and hence determine palindrome-ness */

/* print out an appropriate message */
```

3. **Compile and Debug:** Compile and debug your program until it compiles without errors.

4. **Execute:** Run your program and verify that it runs correctly. Then, make a script file with a cat of your program and three runs - illustrating that the program works. Call the script file `palindrome.scr`.

10.5 What to Hand In

Hand in the script files from above, stapled as a package - with your name and section number clearly indicated on the front page.

Chapter 11

Lab - Call-by-value and Call-by-address

11.1 Goals

This lab is an exercise to familiarize you with the following:

- Using pointers to accomplish call-by-address parameter passing.
- Call-by-value versus call-by-address parameter passing.

11.2 Reference Materials

Chapter 7 of Deitel.

11.3 Call-by-Value versus Call-by-Address

This lab demonstrates that a calling routine can pass its actual parameters to a function's formal parameters in one of two ways: call-by-value or call-by-address. Using call-by-value, the called function gets a COPY of the actual parameter. Any changes that the function makes are to the COPY; thus the original value of the actual parameter in the calling routine is NEVER changed. (No side effects.) This is what we have seen so far in our passing of parameters (other than arrays) to functions.

Using call-by-address, the function receives an ADDRESS of the actual parameter. The function cannot change the address, but by using indirect addressing, it can modify the

original actual parameter in the calling routine. (Yes, side effects.) Independent of call-by-value or call-by-address, a function can return a value using the `return` statement.

1. Copy the file `swapper.c` into your own home directory. This file contains an incorrect solution to defining a function that takes takes two parameters, and swaps their contents if the first parameter is smaller than the second parameter. Compile and execute the program as it is now, noting that the output is NOT correct.

This solution is incorrect because the programmer (i.e., me) did not take into account that (EXCEPT FOR ARRAYS), by default, C passes actual parameters to a function using call-by-value. Therefore, even though the values of `a` and `b` seem to get swapped within the function `interchange`, in fact, back in the main program, the function call has no effect on `a` and `b`. To obtain the effect of call-by-address parameter passing in C, you need to use pointers. This is explained in Chapter 6 of your textbook.

```
/*********************************************************************
 * Programmer:                                                       *
 * Course: CISC105                                                   *
 * Section:                                                          *
 * Lab Time:                                                         *
 * File: swapper.c                                                   *
 * Date:                                                             *
 * Title: Call by Value versus call by reference                     *
 *                                                                   *
 *********************************************************************
 */

#include <stdio.h>

/*function prototypes*/
int interchange (int, int);

main() {
   int
      a,b,           /* used to test function interchange */
      didSwapOccur;  /* indicator if function did the swap*/

   a = 6;      /* assign arbitrary values in order to test interchange */
   b = 50;

   printf("Main: Before call to function: a = %d  b = %d\n",a,b);
   didSwapOccur = interchange(a,b);
   printf("Main: After call to function:  a = %d  b = %d\n",a,b);

   if (didSwapOccur == 1)
      printf("Main: interchange was made\n");
   else
      printf("Main: no interchange was performed\n");
}
```

```
/*********************************************************************
 * Programmer:                                                       *
 * Function: interchange                                             *
 * Description:                                                      *
 *   This function compares its two formal parameters x and y.  If the *
 *   first is smaller than the second, the values are swapped,       *
 *   otherwise they are left alone.                                  *
 *   Return values:                                                  *
 *      0 if no swap occurs                                          *
 *      1 if swap occurs                                             *
 *********************************************************************/
int interchange(int x, int y) {
   int
      temp;        /* temporary location needed to perform a swap */

   if (x < y) {
      temp = x;    /* x is less than y, so swap the two values    */
      x = y;
      y = temp;
      return (1);
   }
   else
      return (0); /* x is not less than y, so no swap needed     */
}
```

2. Modify the program so that the actual parameters `a` and `b` are passed using call-by-address rather than call-by-value, thus allowing the effects of function `interchange` to be felt by the main program. Be sure to also modify the comments accordingly.

Compile and run the new version of the program. The corrected program's output should look something like:

```
Main: Before call to function: a = 6  b = 50
Main: After call to function:  a = 50  b = 6
Main: interchange was made
```

Create a script file that cat's your modified program, removes a.out, compiles the program, and runs it. Name this script file `swapper.scr`. Print this script file for grading.

11.4 Explicit Return Versus Side Effects

This section of the lab demonstrates that under many circumstances, a function can be accomplished in either of two ways: (1) call-by-value parameters with an explicit `return` value, or (2) call-by-address parameters that change the value of the parameter, and no explicit `return`.

1. To your program file above, add a function definition (and corresponding prototype) that takes an integer call-by-value parameter, and explicitly returns the square of the parameter value. That is, add a function definition called `square_by_value` which can be called by calls of the form:

```
printf("The square of %d is %d\n",num,square_by_value(num));
```

Add an interactive session to get a value from the user for the variable `num`, and a printf of the form above to the main program in order to test your new function.

2. Now, add another function definition (and corresponding prototype) that takes an integer call-by-address parameter, and computes the square of that parameter WITHOUT a return statement. That is, add a function definition called `square_by_address` which can be called by calls of the form:

```
printf("The square of %d ", num);
square_by_address(&num); /*call the function */
printf(" is %d\n",num); /*print the value of the actual parameter after the call*/
```

Add the above sequence of statements to your main program after the statements that call the square_by_value function to test your new function with the same value for `num`.

When you are certain that your new version of the program achieves all of the tasks above, create a script file that cat's this program, removes a.out, compiles this program, and executes it with the input number 9 and then with the number 6. Name this script file `lab10-2.scr`. Print out this script file for grading.

11.5 A Small Practice Program

1. **Problem Statement:** Several 3-dimensional objects will be used in a program: cubes, right circular cylinders, and spheres. For each of these objects we will need to compute the volume and the total surface area - using a single separate function fr each type of object.

2. **Program Development:** Following is an outline for the program; complete the missing C statements and create file called `shapes.c`.

   ```
   /* the usual comment block here....*/



   /* and header files ( don't forget the math library) */
   ```

```
/* function prototypes for 3 functions:
          getCubeInfo
          getSphereInfo
          getCylinderInfo

using call-by-value for the object parameters (eg. radius of sphere),
and call-by-address to return the volume and surface area. */

/* open main */

/* declare variables for radius and height, and volume and surface
   area, and a variable to hold the shape type (char) */


*/ start a while loop (use shape = 'x'  to stop) */

/* make a switch statement to switch on shape type:
   s for sphere, c for cube, and r for cylinder.  for each case,
   prompt for the relevant data, then call the appropriate "get"
   function to compute the volume and surface area, and then output
   the results. */

/* close the loop */

/* close main */
/* put your 3 function definitions here. */
```

3. **Compile and Debug** Compile and debug your program until it compiles without errors.

4. **Execute:** Run the program, and test using different data values, and verify that it runs correctly. Then make a script file called `shapes.scr` in which you cat your program and run it - showing that each case works.

11.6 What to Hand In

Hand in your three script files, stapled as one package, with your name and section clearly indicated on the front page.

Chapter 12

Lab - Iteration and Recursion

12.1 Goals

This lab is an exercise to familiarize you with:

- The Iterative approach to problems.
- The Recursive approach to problems.

12.2 Reference Materials

Chapter 10 of textbook.

12.3 Iteration Versus Recursion

1. Copy the file called `gcds.c` into your own home directory. This file contains the following C program:

```
/*****************************************************************
 * Programmer:                                                   *
 * Course: CISC105                                               *
 * Section:                                                      *
 * Lab Time:                                                     *
 * File: gcds.c                                                  *
 * Date:                                                         *
 * Title: Iteration versus Recursion                             *
 *                                                               *
 * This program shows both the iterative and the recursive methods     *
```

```
 * for solving the same problem.                                          *
 * The problem is to compute the greatest common divisor of two data      *
 * values.  The greatest common divisor is the largest integer that       *
 * will evenly divide both data values.                                   *
 *                                                                        *
 **************************************************************************
 */

#include <stdio.h>

/*function prototypes*/
int iter_gcd(int, int);
int recur_gcd(int, int);

main() {
  /*declaration of variables */
   int num1, num2;  /* two values entered by the user*/

   /*interactive session to input the two numbers for which to determine
   their greatest common divisor*/

   printf("Enter two integers separated by a space: ");
   scanf("%d %d",&num1,&num2);

   /*call iterative version of gcd function to compute greatest common
   divisor of num1 and num2 and print out the answer returned*/
   printf("The greatest common divisor of %d and %d, iteratively computed is: %d\n",
      num1,num2,iter_gcd(num1,num2));

   /*call recursive version of gcd function compute greatest common
   divisor of num1 and num2 and print out the answer returned*/

   printf("The greatest common divisor of %d and %d, recursively computed is: %d\n",
      num1,num2,recur_gcd(num1,num2));
}

/*function definition for iterative computation of greatest common divisor
of two integers*/
int iter_gcd(int n1, int n2) {
  int i;             /* loop index variable*/
  int greatest = 1; /* gcd as it is computed*/
  int smaller;       /* smaller of n1 and n2 values - used for loop bound*/

  if (n1 < n2)
     smaller = n1;
  else
     smaller = n2;

  for (i=2; i<=smaller; i++)
     if (((n1 % i) == 0) && ((n2 % i) == 0))
        greatest = i;
```

```
  return greatest;
}

/*function definition for recursive computation of greatest common divisor
of two integers*/
int recur_gcd(int n1, int n2) {
  int greatest;     /*gcd as it is computed*/

  if (n2 == 0)
     greatest = n1;
  else
     greatest = recur_gcd(n2,n1 % n2);
  return greatest;
}
```

2. Compile this program, and execute it with input pairs 24 and 36, 54 and 36, 25 and 64.

3. Copy this program into a new file, and modify this new file to use a TRACE_ITER flag to turn on tracing of the iterative approach to this problem, just as we used the DEBUG flags to trace for debugging. Turn on this flag with a `#define TRACE_ITER 1` at the top of your file. Then, insert `IF (TRACE_ITER) printf...` style statements appropriately inside the inter_gcd function to trace which values are being checked for being possible candidates for the greatest common divisor of the two data values, and to check the current greatest common divisor as the loop executes. Compile this version of the program and execute it with the input value pairs above. When you are sure that it is tracing the correct information, create a script file that cat's this program, removes a.out, compiles this program, and executes it with the pair 24 and 36. Print out this script for grading.

4. Now, examine the output of the traced execution of your program, and the actual code of the iter_gcd function to understand exactly how this approach works. On a separate piece of paper (or in a separate file), write a paragraph in English sentence form that explains the iterative approach to computing the gcd. You should NOT just write out each statement in the iterative gcd function in English, but rather summarize the approach in a few sentences. You will hand this in as part of your lab assignment.

5. Look more closely at how the iterative solution is working, and rethink the goal of computing the gcd of two numbers. This version of the iterative approach is doing more work than necessary. Copy this program to a new file called gcds-new.c, and modify the iterative part of this program to make it more efficient in its determination of the gcd. Hint: The loop is supposed to find the GREATEST common divisor. The program would be more efficient if the loop was set up to take advantage that it is looking for the GREATEST common divisor, and once it is found, it can quit iterating.

To verify that your new version does less work at execution time, insert code into BOTH versions to count the number of times that the loop is executed. You will need a new

counter variable, and statements to initialize it before the loop, and then increment it inside the loop, and print the final counter value after the loop. When you are sure that these programs work properly and that your counting is working ok, edit the program files to TURN OFF the tracing, but leave in the counting, create a script file that cat's your more efficient version of the program. With the script still recording what you do, perform the following steps: Compile your revised version that contains the more efficient iterative version and execute it with the value pairs above. Write down the printed iteration counts in a table (see below). Compile the old version that contains the less efficient iterative version, but performs the counting of iterations, execute it with the same data pairs, and write down these iteration counts in the same table. Describe why the counts are different and why your second version is always the same or better in efficiency, but will never be worse in efficiency. Print out this script file for grading. This table will also be included for grading.

Your table should look like:

```
Data Values          Original Version          New Version
                      Iteration Count          Iteration Count
-------------------------------------------------------------
24, 36
54, 36
25, 64
```

6. Now, starting with your original file of the program (gcds.c) that has no counting, copy this file into a new file, and add tracing for the recursive approach to the problem. That is, create a `TRACE_RECUR` symbolic constant by `#define TRACE_RECUR 1` at the top of your program file, and add `if (TRACE_RECUR) printf...` style statements into the `recur_gcd` function to trace its execution. For each recursive call to `recur_gcd`, you should print out the current values of the two parameters and the computed value for greatest just before returning. Each call should result in one line of output. Nicely label each line of output so the user can clearly see what is happening in the recursion. When you believe that you have achieved a nice tracing capability, create a script file that cat's this version of your program, compiles it, and executes it with the data pair 24 and 36 only. Print out this script file for grading.

7. On a separate piece of paper or in a separate file, write a paragraph in English sentence form that explains the recursive approach to computing the gcd. You should NOT just write out each statement in the recursive gcd function in English, but rather summarize the approach in a few sentences. You will hand this in as part of your lab assignment.

12.4 What to Hand In

Please submit the following items IN THIS ORDER, stapled together as one package, with your name and section clearly indicated on the front page.

1. Script file of trace of the original iterative version of gcd.
2. English explanation of iterative solution.
3. Script file for two different versions of iterative solution with counting, but no tracing.
4. Table of iteration counts from the two different iterative solutions.
5. Script file for trace of recursive version.
6. English explanation of recursive solution.

Chapter 13

Lab - Pointers and Linked Lists

13.1 Goals

This lab is an exercise to familiarize you with the following:

- Using pointers to create and manipulate dynamic data structures.
- Linked Lists.

13.2 Reference Materials

Chapter 12, pp. 450-501 of Deitel.

13.3 Creating and Manipulating a Linked List with Pointers

1. Copy the file `lists.c` into your own home directory. Compile and run this program, choosing the first option to add new items to the dynamically growing list and printing out the list by choosing the third option.

```
/*************************************************************************
 * Programmer:                                                           *
 * Course: CISC105                                                       *
 * Section:                                                              *
 * Lab Time:                                                             *
 * File: lists.c                                                         *
 * Date:                                                                 *
 * Title: Pointers for Linked Lists                                      *
 *                                                                       *
```

```
 *                                                                      *
 ************************************************************************
 */

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

/*function prototypes*/
void display_menu();

main() {

struct listnode  {   /*declaration of linked list element*/
  int data;
  struct listnode *nextptr;
  } *currentptr, *list;  /*pointer variables for using list*/

int choice;   /*menu choice*/
int item;     /*data item to add to list*/

currentptr = NULL;
list = NULL;
display_menu();
printf("Choice? ");
scanf("%d",&choice);

while (choice != 0)  /*type 0 to quit*/
   {
   switch (choice) {
      case 1: /*insert into front of list*/
              printf("Enter a number:");
              scanf("%d",&item);
              currentptr = list;
              list = malloc(sizeof(struct listnode));
              list->data = item;
              list->nextptr = currentptr;
              break;
      case 2: /*delete first element*/
              break;
      case 3: /*print out the contents of the list*/
             if (list == NULL)
                 printf("List is empty.\n");
              else
                 {
                 printf("The list is:\n");
                 currentptr = list;
                 while (currentptr != NULL)
                    {
                    printf("%d --> ",currentptr->data);
                    currentptr = currentptr->nextptr;
                    }
```

```
                printf(" NULL\n\n");
                }
            break;
      default:
            printf("Invalid choice. Try again.\n");
            break;
      }
   printf("Choice? ");
   scanf("%d",&choice);
   }

printf("Have a good day!\n");
}

/*function to display menu of choices*/
void display_menu() {
   printf("Here are your choices:\n");
   printf("   1: Insert an element to the front of the list.\n");
   printf("   2: Delete the first element of the list.\n");
   printf("   3: Display the whole list.\n");
   printf("   0: quit.\n");
}
```

2. Note that the code for deleting an item from the list is not currently present, so that if you try the second option when running the program, it will not delete anything from the list. Add the necessary code to this case of the switch statement to delete the first item in the list when the user chooses the second option. Be sure to free up the space that was used by the deleted item. Also, make sure that your program does NOT try to delete an item from an empty list, but instead displays a message that tells the user that the list is currently empty, and thus nothing is deleted. Compile and run your new version, adding some items first, and then deleting several items to check your code.

3. Now, add "option 4" to the menu display to allow the user to request that the value of every item in the list be doubled. That is, if the list currently looks like:

```
2 --> 10 --> 5 --> NULL
```

Then, by the user typing 4 as their choice from the menu, the list will become:

```
4 --> 20 --> 10 --> NULL
```

Option 4 should also display the new list after the doubling has occurred to show the user that the list has been changed.

13.4 What to Hand In

When you are certain that your new version of the program achieves all of the tasks above, create a script file that cat's this program, removes a.out, compiles this program, and executes it. During execution, you should add several items, print out the list, delete several items, print out the list, add some more items, print out the list, request the doubling option, and then delete items until the list becomes empty, print out the list, and lastly try to delete the last item. Print out this script file for grading.

Part III

Programming Projects

Chapter 14

General Guidelines for All Programs

14.1 Program Style

Approximately 20% of the grade depends on your program's readability. Your program style SHOULD conform with the example programs shown in the textbook.

Style considerations include, but are not limited to:

- meaningfulness of the variable names
- indentation and white space used to improve readability of your program
- appropriate use of comments including: description of algorithm used, description of constants and variables, description of inputs and outputs

14.2 Submission Instructions

- All computer output must be printed by a laser printer; all additional written material must be on 8-1/2" x 11" size paper.
- When output consists of more than one sheet, the sheets must be submitted in correct sequence, and stapled together.
- Fill out and staple the cover sheet to the front of the program. This cover sheet is online in the file `Progs/coversheet.`
- Programs may be submitted up to 5 days after the due date.

- REMEMBER! Programs will not be accepted beyond the Latest Submission Date without approval of the Professor. Such approval will be granted only under special circumstances such as documented illness.

14.3 Summary of C Coding Style Guidelines

- BEGIN EVERY PROGRAM WITH A BLOCK OF COMMENTS.
- PLACE COMMENTS WITHIN CODE EITHER IN THE SAME COLUMN AND ABOVE THE CODE IT EXPLAINS, OR TO THE RIGHT OF THE CODE.
- A COMMENT IS OF NEGATIVE VALUE IF IT IS INCORRECT.
- CHOOSE MEANINGFUL VARIABLE AND FUNCTION NAMES; DO NOT OVER-ABBREVIATE.
- EACH INDENTATION SHOULD BE 3 SPACES; NOT 2, NOT 5, ALWAYS 3.
- ALIGN CORRESPONDING { }'S; INDENT CODE WITHIN BOUNDING { } 3 SPACES.
- If STATEMENTS: ALWAYS INDENT THE then PART UNDER THE if PART.
- If-then-else STATEMENTS: ALIGN THE KEYWORDS if AND else, AND ALIGN THE then AND else CODE.
- Switch STATEMENT: INDENT EACH CASE'S CODE.
- For AND while STATEMENTS: ALWAYS INDENT THE BODY OF THE LOOP.
- KEEP CODE READABLE: READABLE DOES NOT MEAN EXECUTES FASTEST.

14.4 A Note on These Projects

The following chapters are *example* program projects. Some of these might be replaced by your instructor. You are encouraged to work through as many of these projects as possible, even if they are not explicitly assigned.

14.5 Getting Help

IF you are having some problem with your attempts at a program for CISC 105 Lab or Project, it is a good idea to either email a copy (with explanation) to the TA and/or have a session at his/her terminal so he/she can see what's actually in your program. This will work much better than merely telling us about your bug. There are so many ways a program can go wrong, it helps if the TA can SEE (and maybe run) your program so he/she can advise you, help you find/correct bugs, etc.

14.6 Project Coversheet

When turning in a programming project, always include a coversheet - available online, and illustrated below. Always fill in this sheet and staple to the front of your project.

```
            CIS 105 Programming Assignment


     Assignment:  Project #
     Course:
     Course Section #:
     Programmer:
     Login name:
```

CORRECTNESS	Possible	Grade
C: Correct operation	30%	
T: Testing & Error Handling	15%	
CLARITY		
D: Documentation	15%	
C: Code structure	10%	
N: Naming	10%	
F: Formatting	10%	
SUBTOTAL	90%	
A: A Grade	10%	
TOTAL	100%	

Chapter 15

Project 1 - The Company Payroll

[1]

15.1 Objectives

The goal of this project is for you to learn to develop a complete C program from scratch.

- Develop an algorithm and plan a program based on the algorithm
- Write a C program based the algorithm
- Identify and correct syntax errors
- Test a program to see if it contains logic errors.

Required Background: **Before** you begin this project, be sure you have read chapters 1-4 of Deitel.

15.2 Assignment

The **Titanium Manufacturing Firm** has just hired you to write the software for their payroll system. They want the program to perform the following actions:

1. Print an informative, but short, message to the payroll clerk as they begin to execute the program.

[1] This programming project is based on problem 4.28 in Deitel.

2. Ask the clerk if they are ready for their first employee, and wait for the clerk to type 1 for yes or 0 for no. If they type 0, end the execution; otherwise, begin to input data about the first employee's pay check (described below). After the first employee has been processed, then ask the clerk whether they want to process another employee. If they type 0, end the execution. If they type 1, repeat the processing for this new employee. The processing of employees should continue to repeat until the clerk types 0 in response to being asked whether they want to process a new employee. Therefore, the program should handle any number of employees, maybe 3 on one execution, and 100 on another execution.

3. For each employee, interactively obtain the following information from the clerk, by a nice, user-friendly interactive session:

   (a) the id number of this employee

   (b) the status - H for hourly, S for salaried staff, C for contractor

   (c) If the status is H, the hours worked and the hourly payrate

   (d) If the status is S, the annual salary

   (e) If the status is C, the amount to be paid

4. After this information is all input by the clerk in response to your program prompting them for it, then compute the employee's paycheck. The display the check information on the screen.

5. The amount of the check is computed as follows - based on the value of status.

   (a) H - compute hours times rate (with time and one half for overtime hours (hours over 40))

   (b) S - Take annual salary divided by 52

   (c) C - the check amount is the amount of the contract

   For example, an employee with 'H' status with 50 hours of work, and a rate of 12 dollars per hour would earn 660 dollars.

6. Print out a "check" to the screen. so that it looks like a check.

7. After all employees have been processed, print a friendly message to end the day for the clerk.

When you are ready to type in your program, enter it into the file `payroll.c` and start editing with this file.

15.3 What To Hand In

Note: Remember to follow the programming guidelines listed in the beginning of the Program Projects section of this manual.

Run your program several times and satisfy yourself that it produces correct answers. Be sure to try typical values for the answers to the program's questions, as well as legal, but, what we call boundary values, such as 0 and 1 for some of the answers. For example, try 0 for number of hours worked, and make sure you print an appropriate message instead of a check.

When you are sure your program works, make a script file by doing the following:

```
script payroll.scr
cat payroll.c
cc payroll.c
a.out
exit
```

Show executions with the following input cases:
(1) only one employee
(2) no employees
(3) 2 employees of each status type

Hand in a copy of your script file printed on a laser printer.

15.4 To Earn an "A" Grade

The "A" task is to add the following features.

1. Extend it so you handle erroneous input nicely. That is, whenever the clerk inputs an invalid number for any of the questions, your program should tell the user that the input is wrong and repetitively prompt for a legal value until they input a legal value. Hint: This will involve adding a WHILE loop around the code that interactively inputs a value.

2. Extend the program to display a total number of employees processed and the total amount of checks for the day for each status type after the last check is printed.

You can assume that the clerk inputs the correct type for each question, that is, integers, not letters or decimal numbers. You should check to be sure that the input is not negative

number for the id number, hours worked, or any other numeric input, and that the status is a valid letter.

You should also make sure that the clerk inputs either 0 or 1 in response to whether they want to process another employee.

Chapter 16

Project 2 - ATM Machine Software

16.1 Goals

To write a simulation of an ATM machine, and practice th following programming techniques.[1]

- Develop an algorithm and plan a program based on the algorithm
- Implement a C program based the algorithm
- Identify and correct syntax errors
- Test a program to see if it contains logic errors.
- Use the following features of C programming:
  - Character and integer input/output.
  - Nested control structures.
  - Definitions of simple functions.
  - Use of global variable.
  - Checking for bad input, and reporting errors.

16.2 References

You should have completed the first 7 labs, and have read chapters 1 - 5 of Deitel before beginning this program.

[1] This programming project was inspired by Mike Matsumoto - former CIS student who passed away in 2000.

16.3 Assignment: ATM Software

16.3.1 Overview

We will write a simulation for a *simple* ATM machine - one that has a single account, and only one user. Also, the account will always start with a zero balance. Even with this very simple version, the program will have many of the functions of a *real* ATM program - remember that these machines actually are controlled by software (and that the software is probably written in C).

16.3.2 Program Specifications

Your program will start up by displaying the following menu:

```
 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$                                     $
$    WELCOME TO MUCHO DINERO BANK     $
$                                     $
 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

       Enter 1 To Open an Account
       Enter 2 To Close an Account
       Enter 3 To Deposit Money
       Enter 4 To Withdraw Money
       Enter 5 To Get Current Balance
       Enter 6 To Set PIN
       Enter 7 To Quit
       Enter 8 To Redraw Menu
```

Use a loop to prompt for and scan in a selection. After a selection has been made, a *switch* statement will be used to match the input with the correct function. The details of each function are described below.

Your program should define the following global variables:

```
float balance;  /* balance of the account */
int pin; /* pin number of the account */
int account_active; /* set to 1 if there is an active account, 0
     otherwise */
```

16.3.3 Required Functions

Your program should include the following functions:

1. `void display_menu(void);`

   Prints a menu of items for the user to choose from.

2. `void hey(void);`

   Prints "Have a nice day too!" message. (Especially important in New York).

3. `void open_account(void);`

   There can be only one account while the program is running. If there is not already an account opened, this function opens an account and sets the balance to 0.0 and the PIN number to default of 999. If there already is an opened account, the function prints an error message.

4. `void close_account(void);`

   Closes the account if an open account exists (if the PIN is entered correctly). Otherwise, prints an error message.

5. `void deposit(void);`

   If there is an open account, this function deposits money in the account (balance is incremented by the entered amount). If the amount entered is zero or less, an error message is printed. No need to check the PIN to deposit money.

6. `void withdraw(void);`

   Withdraws money from the account if there is an open account and the correct PIN is entered; prints error if not. If the amount entered is zero or less OR if there is insufficient funds, an error message is printed.

7. `void get_balance(void);`

   Prints the current balance if the account is active and the correct PIN is entered, prints error if not.

8. `void set_pin(void);`

   Changes the current PIN number if the account is active and the correct old PIN is entered. It asks for the new PIN twice for confirmation. An error message is printed if PINs do not match.

9. `int get_pin(void);`

   Prompts for PIN and returns 1 of the correct PIN is entered, 0 otherwise. This is a utility function for the other functions to use. It should NOT be called from the main() function.

16.3.4 Error Handling:

If, at any time, the user enters an inappropriate value (wrong PIN, invalid selection, etc.), an error message will be printed. A sample implementation will be provided in your account. You should check this executable program for the details of your error messages.

16.4 Helpful Hints

A skeleton program is available as `atm_skeleton.c` in the `Progs` directory. You should provide the missing parts in the skeleton. If you compile the skeleton program, you will see that the options 1, 5, 7 and 8 are already functioning.

Remember to save your programs regularly (every 30 minutes or so).

After fixing all compiler errors, you will want to run your program in such a way as to exercise **ALL** of your code. Try various error situations, to be sure that your program handles theses correctly.

16.5 What To Submit

Run your program several times and satisfy yourself that it produces correct answers in all cases! When you are sure that you program works, make a script file for grading:

```
script atm.scr
cat  <yourfilename>.c
cc <yourfilename>.c
a.out
   ...
   various options....
exit
```

Hand in a copy of your script file.

16.6 What to do for an "A"

Change the menu options so that the user enters a letter instead of a number to choose a selection. (Eg. 'W' for a withdrawal, 'D' for a deposit, etc.) Then use the `getchar()` function from `stdio.h` to scan in the user's selection.

Add two functions as follows.

1. `char Get_Command()`

   Prompt and read in the user command - and return it.

2. `int Verify_Command(char Cmd)`

   Determine whether (1) or not (0) `Cmd` is a valid command.

Change the loop in main to take advantage of these new functions.

Make a script file as in the regular part, and hand in just this scrit file.

Chapter 17

Project 2 - The Hangman Game

17.1 Objectives

- To learn to use simple character i/o
- To design software that uses strings
- To use global variables
- To design to specifications

Background Reading: Make sure you have read the following chapters in Deitel: Chpater 4 (control structures), Chapter 5 (functions), and Chapter 8 (strings); and completed the chapters 1-7 in the lab manual before beginning this project.

17.2 Overview

You have been hired by the *ABC Gaming Company* to design game software. As a test of your design and programming skills, they have given you a simple first assignment — to design and program part of the *Hangman* game. Recall that *Hangman* is a simple word-guessing game, where the player guesses words one letter at a time. To keep things simple, we'll just use 4-letter words, and the little hangman figure has only four *states* — see the sample run.

17.3 Specifications

The hangman program will display the following on start-up:

```
     USED LETTERS:

         -------
         | /   |
         |/
         |
         |
         |
         |
     |--------------|

     YOUR WORD: ____

     GUESS A LETTER:
```

After guessing a correct letter (say u for luck):

```
GOOD GUESS!
USED LETTERS: u

         -------
         | /   |
         |/
         |
         |
         |
         |
       |--------------|

YOUR WORD: _u__

GUESS A LETTER:
```

After guessing an incorrect letter (say i):

```
        i WAS AN INCORRECT GUESS
        USED LETTERS: ui

            -------
            | /   |
            |/    O
            |
```

```
      |
      |
      |
|--------------|

YOUR WORD: _u__

GUESS A LETTER:
```

The game will end after either correctly guessing the word, or after four wrong guesses.

17.4 Getting Started

A skeleton program is available in `Progs/hang_skel.c`, and a running program is in file `Progs/hangman`. Also, the dictionary is in file `Progs/Words`. You should copy or download these files to your Unix account. Make sure that the file `hangman` is executable by entering the command `chmod +x hangman` — you can then run the program to see a sample run.

Your program will use the following global variables:

```
char dictionary[MAX_WORDS][MAX_LENGTH+1];  /* 1000 word dictionary */
char used_letters[26]; /* array for used letter */

/* the length of dictionary (don't touch this variable!!!) */
int total_words;
```

Your program will use the following functions:

1. `int load_dictionary(char *filename, int no_words)`

   This function loads the dictionary array from the Words file. This function is provided for you.

2. `char *get_random_word()`

   This function gets a random number to be used to select a new guess every time the game is played. The function returns the word to be used next. This function is provided for you.

17.5 What To Hand In

Please make sure that the base and the hanging man displays are EXACTLY as they appear in the sample program. There will be points deducted for cosmetic mistakes. You should be able to display outputs matching EXACTLY to the specification.

Hand in a script file with a listing of your program, and a run illustrating all of the possible options.

17.6 Skeleton Starter Program

```
/* File: hang_skel.c */
#include <stdio.h>
/* constants: */
#define MAX_WORDS 1000
#define MAX_GUESSES 4
#define MAX_LENGTH  4

/* prototypes: */
void display_base(int);
int load_dictionary(char *, int);
char *get_random_word();

/* globals: */
char dictionary[MAX_WORDS][MAX_LENGTH + 1];
char used_letters[26];
int total_words;

int main() {
   char *filename = "Words";
   char input;
   int wins, losses;
   int done = 0;
   int i;

   char word[MAX_LENGTH+1], guess[MAX_LENGTH+1];

   int word_length;
   int wrong_guesses;
   int right_guesses;
   int total_guesses;
   int letters;

   /* set the randon number generator */
   srandom(time(0));

   /* load the dictionary file */
```

```
   total_words = load_dictionary(filename, MAX_WORDS);

   if (total_words != -1)
    printf("Loaded %d words from %s\n", total_words,
    filename);
   else {
    printf("Error loading %s\n", filename);
    return;
   }

   /* while loop for multiple games */
   while(done == 0) {
      total_guesses = 0;
      wrong_guesses = 0;
      right_guesses = 0;
      used_letters[0] = '\0';

      strcpy(word, get_random_word());

      printf("WORD: %s (given away to test not to cheat)\n",
    word);

/* make sure to take this line out of here afterwards */
printf("\n\n *** Type Control C to get out *** \n");

      word_length = strlen(word);

      /* initialize guess to:  ____ */

      for(i = 0; i < word_length; i++) {
    guess[i] = '_';
      }
      guess[i] = '\0';

      /* put 9 \n's to clear the screen */

      /* while loop for a single game */
      while(wrong_guesses < MAX_GUESSES &&
        right_guesses < word_length) {
           /* based on the previous status, print:

    USED LETTERS:

                    -------
                    | /   |
                    |/
        |
                    |
                    |
                    |
                    |--------------|
```

```
       or with the hanging man.
       and then print:

    GUESS A LETTER:

    */

    /* put 9 \n's to clear the screen */

    /* based on input, print:
    c IS AN INVALID CHARACTER
       or
    YOU HAVE ALREADY USED c
       or
    GOOD GUESS!
       or
    c WAS AN INCORRECT GUESS
    */

   } /* end while for a single game */

   /* based on the result of the game, print:
    YOUR WORD: luck
    YOU LOSE
      or
    YOUR WORD: luck
    YOU WIN!
      and then
    SO FAR, WINS: x  LOSSES: y
    PLAY AGAIN (y/n)?
   */

  } /* end while loop for multiple games */
} /* end main */


void display_base(int status) {
   /* display the hangman based on the number
      of wrong guesses made so far; */

}
```

17.7 What to do for an "A"

For an "A" do each of the following.

1. Change the number of states for the hangman - from 4 to 6 by drawing legs and arms one at a time.

2. Change the maximum word length to 6, and adjust all your code - so that it correctly deals with a variable length for word length.

3. Then edit the words file and add some words of mength 5 and 6.

Chapter 18

Project 3 - The Data Processing Assistant

18.1 Objectives

- To write functions which have one-dimensional arrays as arguments
- To implement an algorithm which works with array data
- To write functions which interface with existing code

Required Background: **Before** you begin this project be sure you have read chapters 6 nd 7 of Deitel, and that you have completed the lab on one-dimensional arrays.

18.2 Part I (60 points) - Sorting and Merging

For this part of the program, you must write two general functions: one that sorts an array into ascending order, and one that merges two arrays (assumed to already be sorted) into one sorted array using a merge algorithm to be discussed in class.

A template containing a main program is given to you in: `merge-template.c`. Your task is to code the missing functions called SortArray and Merge. (No changes are permitted to the main function other than added appropriate header comments.)

In this template, the main program inputs two sets of numbers. (For sake of example, let's assume the numbers represent weights of certain animals at the Philadelphia Zoo.) For each set of animal weights, the main program (1) inputs the number of weights in the set, (2) inputs the weights, and (3) calls your sort function to sort the set of weights.

The main program then calls your merge function to merge together the two sets of weights.

Example input:

```
6
6 8 9 3 15 11
8
6 22 9 10 4 15 22 12
```

The final output will indicate the sorted and merged array values:

```
3 4 6 6 8 9 9 10 11 12 15 15 22 22
```

Notice that there may be duplicate values within each array and between arrays. All of these duplicate values should appear in the merged array.

Test this program before going on to Part II.

18.3 Part II (30 points) - Input/Output Functions

For this part of the program, you will add two new functions as follows; and, you are allowed to make small modifications to the main function - so that it uses these new functions.

1. `void printArray(char* message, int x[], int n);` - a function whose job is to print the message on one line, then the array x of n values on the next line...then print a blank line at the end.
2. `int inputArray( char* promptMessage, int x[], int MaxSize, int* nValues);` - a function whose job is to print the prompt message, and then read in the number of data, and then use a loop to read in the array elements. Use the return value of the function to return 1 if the function worked without errors, 2 if the number of data exceeds the size of the array, or is less than 1and return a 3 if the user enters insufficient numbers.

   The main function uses **`inputArray`**, sending it an appropriate prompt message, the array name, the declared array size, and a variable to hold the count (dont't forget the &).

Once you have written thses functions, make a **new** copy of your program and enter your functions at the end. Add function prototypes, and then replace the original code for input - with calls to your **`inputArray`** function. Then add calls to function **`printArray`** as follows:

```
....

printArray("first array - unsorted", ... );
```

```
...

printArray("fisrt array - sorted", ... );


... do the same for the second array ...


... then output the merged array using printArray
```

Once the program is debugged, test it with the test data files. Then make a script file as shown below.

18.4 What To Hand In

Run your program 5 times with 5 test data sets that will be made available a few days prior to the due date. You should be POSITIVE that your program operates correctly BEFORE running it with this test data. Make up your own test data to help debug and verify your program.

Even if your program gets correct results for my test data, if your program contains logic errors that would be uncovered by other test data, you can lose significant credit. Do NOT wait until a few days prior to the due date to begin writing or testing your program!!

Your actual script execution will look like:

```
script scriptfilename
rm a.out
cat  your_merge_program.c
cc   your_merge_program.c

cat  Merge_Program.test.data1
a.out < Merge_Program.test.data1

cat Merge_Program.test.data2
a.out < Merge_Program.test.data2

cat Merge_Program.test.data3
a.out < Merge_Program.test.data3

cat Merge_Program.test.data4
a.out < Merge_Program.test.data4
```

```
cat Merge_Program.test.data5
a.out < Merge_Program.test.data5

exit
```

Make another script file for your modified program from section 3 - using the same runs and the same inout data.

18.5 Notes

1. Your functions SortArray and Merge should be GENERAL, that is, written to sort any array of integers, and to merge any 2 arrays of integers, respectively. They should not be specific to sorting or merging animal weights, hence the variable names used in your functions should be general, not for example, 'AnimalWeights1[]', 'AnimalWeights2[]'
2. You may NOT change the main program to merge the two unsorted arrays into one large unsorted array, and then sort the merged array.
3. Even when you add your calls to `inputArray` and `printArray`, do NOT alter the calls to sort and merge.

18.6 To Earn an "A" Grade

Change your merge function such that it removes duplicates; that is, the merged array returned should contain NO duplicate values.

Make up your own test data case to CLEARLY demonstrate that your new program works.

Chapter 19

A Tic Tac Toe Program

19.1 Objectives

- Develop an algorithm and plan a program based on the algorithm
- Implement a C program based the algorithm
- Identify and correct syntax errors
- Test a program to see if it contains logic errors
- Write an interactive game program
- Use the following features of C programming:
  - Function definitions and calls
  - Passing Parameters and returning values from functions
  - Using 2 dimensional arrays

Reguired Background: **Before** you begin this project be sure you have read the chapters in Deitel on arrays and functions, and that you have completed **both** labs on arrays.

19.2 Assignment: A Tic Tac Toe Program

19.2.1 Overview:

You have all probably played a computer game at sometime in your life. Now, you have the opportunity to create your own program that allows the user to play tic tac toe against the computer, (ie., your computer program!).

For those not familiar with tic tac toe, or those who forget how to play, it is a board game, with a 3 by 3 board that looks like:

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
```

The game has two players, one designated the X player, and one designated the 0 player. Each player takes turns placing their mark, X or 0, on the board in a square that has not been marked yet. The objective of the game is to be the first player to have 3 marks in a row, column, or diagonal.

Your program will be written so that the person who runs your program is the X player, and they take turns with the computer, which is the 0 player. Your program will allow the user to play one game, and will quit when either the user or computer wins, or all of the spaces of the board have been filled, and noone has won (a tie!).

19.2.2 Functionality of Your Program:

Start by copying the file `tictactoe.c` to your own directory, and starting to work in this file. In particular, your program should accomplish the following tasks:

1. On the screen, display a welcome message and a short (a few lines) explanation of the rules of tic tac toe, and instructions for the user in giving their input.

2. Declare a 2 dimensional array of single characters to store the current values stored in each location of the tic tac toe board. A value will be either 'X', 'O', or ' '.

3. Define and call a function that takes the array as a parameter and clears the board, by initializing each location to contain a blank character ' '.

4. Define a function that draws the board with its current values. For example, a board in which board[0][0] is 'X', board[0][1] is 'O', board[0][2] is 'O', board[1][0] is ' ', board[1][1] is 'X', board[1][2] is 'O', board[2][0] is ' ', board[2][1] is ' ', board[2][2] is 'X', could be displayed as:

```
-------------
| X | O | O |
-------------
|   | X | O |
-------------
|   |   | X |
-------------
```

5. Display the current board to the user, and prompt them to input their desired position for a mark, by inputting a row and column value. Continue to prompt them for a row and column until the coordinates are within the bounds of the board (i.e., within the limits of the array). When they have given a legal position, put their mark, 'X' in that position of the array representing the board. You do NOT have to check to see whether their mark is on top of an 'X' or 'O'; you can assume that the user will always pick a square that is unoccupied. This extra check is part of the extra credit.

6. Define a function that takes the board as a parameter and checks to see whether there is a winner, given the current board values, and returns the value 1 if the current board has a winner, else returns 0 (there is no winner yet). Your program should call this function after the user inputs their desired position, and after the computer puts a mark on the board. The example above shows a situation in which the user has won with a diagonal of 'X's.

7. If the user has not yet won, your program should call the function PutComputerMark() which is already written for you and part of the starter file `tictactoe.c`. After the computer marks the board, you should again check whether the computer has won. Your program should continue to let the user and computer take turns marking the board until either one of them wins, or the board is filled with X's and O's.

8. When there is a winner, your program should print the appropriate message to the user indicating the winner, and exit with a "Game Over" message.

19.3 Required Functions

Your program should at least contain the following functions as separate functions and appropriate calls. You are free to have additional functions defined and used if appropriate.

- `ExplainRules`, a function that explains tic tac toe and gives the instructions to the user on how to play.
- `DrawBoard`, a function that draws the current board.
- `ClearBoard`, a function that clears the board contents, ie., sets each value to ' '.
- `PutPlayerMark`, a function that prompts the user for their position, continues to prompt until legal coordinates, and then places the value in the position.
- `CheckForWinner`, a function that checks the current board for a winner and returns either 1 or 0.

19.4 What To Hand In

Create a script file that cat's your program file, removes a.out, compiles your program, and then runs it 3 times, each time with different inputs from the user, trying to beat the computer. Show an instance where the user wins, one where the computer wins, and one where there is a tie (i.e., the board get filled without a winner).

19.5 To Earn an "A" Grade

1. Add code to the function PutPlayerMark to check that the user has not chosen a position that contains an 'X' or 'O', and if so, continues to prompt them for a new position. Create a script file that cat's this version of your program, compiles it, and runs it showing several attempts to put an X in positions that already have a mark.

2. Add code to the part of the code in the PutComputerMark function that tries an offensive move when a defensive move is not needed, and when a win is not inevitable. Before this code add code that takes a better attack at winning, in particular, replace it with code that will attempt to find a 'O' in a row, column, or diagonal with no 'X' in that row, column, or diagonal, and then place the 'O' in one of the other empty positions in that row, column, or diagonal. When this check shows that this situation does not arise currently, place the 'O' in the manner it is placed now in the code.

Part IV

Appendices

Appendix A

Using Your Home Computer for CISC 105

A.1 A Brief Introduction To Telnet And Ftp

All the assignments and labs for CISC105 must be done on strauss. If you want to work from your home PC (WITH CONNECTION to the Internet) you will need to use the following two tools (if your computer is a MAC, then the MAC clicks to do the following will be a bit different):

1. Telnet

   You need to open the Run window and type the following in it:

   ```
   telnet strauss.udel.edu
   ```

   This will open up another window for you and you will be prompted for your strauss user name and password. Once you have entered them you will see the usual unix prompt in the window. You are now ready to work on your assignments. To exit the telnet window, type the word exit at the unix prompt.

2. FTP (File Transfer Protocol)

   This is a tool you would use to transfer files between two machines. FOR EXAMPLE, suppose all your code etc. is on strauss, and, lets say, you want to print your script file (for submission) out on your printer at home you would first have to transfer the .scr file from strauss to your home PC. To do this you should do the following :

   (a) Open an MS-DOS window either by typing "cmd" or "command" in the Run window or by clicking on the DOS Prompt icon. When the window opens up, note which directory you are in! Change it IF you want to be in a different directory.

   (b) type

```
ftp strauss.udel.edu
```

at the DOS prompt. You will then be prompted for your strauss user name and password. You can then use the "get" command to transfer your file from strauss to your PC. You would type

```
get lab2.scr
```

(c) The file lab2.scr will be transferred to your machine. Then you may exit the ftp session by typing the word "bye" at the ftp prompt.

A typical ftp session would be as follows :

```
ori~/ 31> ftp strauss.udel.edu
Connected to strauss.udel.edu.
220 strauss.udel.edu FTP server (Version wu-2.4.2-academ[BETA-18-VR13](1) Wed Feb 10 10:3
Name (strauss.udel.edu:redeyvai): vaibhavi
331 Password required for vaibhavi.
Password:
230 User vaibhavi logged in.
ftp> ls *.scr
200 PORT command successful.
150 Opening ASCII mode data connection for file list.
file_to_get.scr
226 Transfer complete.
remote: *.scr
15 bytes received in 0.0034 seconds (4.35 Kbytes/s)
ftp> get file_to_get.scr
200 PORT command successful.
150 Opening ASCII mode data connection for file_to_get.scr (83 bytes).
226 Transfer complete.
local: file_to_get.scr remote: file_to_get.scr
87 bytes received in 0.023 seconds (3.69 Kbytes/s)
ftp> put file_to_put.scr
200 PORT command successful.
150 Opening ASCII mode data connection for file_to_put.scr.
226 Transfer complete.
local: file_to_put.scr remote: file_to_put.scr
78 bytes sent in 0.00037 seconds (205.32 Kbytes/s)
ftp> ls *.scr
200 PORT command successful.
150 Opening ASCII mode data connection for file list.
file_to_get.scr
file_to_put.scr
226 Transfer complete.
remote: *.scr
30 bytes received in 0.0028 seconds (10.50 Kbytes/s)
ftp> bye
```

```
221-You have transferred 161 bytes in 2 files.
221-Total traffic for this session was 1063 bytes in 4 transfers.
221-Thank you for using the FTP service on strauss.udel.edu.
221 Goodbye.
ori~/ 32>
```

A command such as ls, when issued at the ftp prompt will list the contents of the directory you are currently in on strauss. The "get" command copies a file from strauss to your PC, "put" will put a file from your PC onto strauss. IF you want to be in a different directory on strauss, change directory appropriately.

Finally, to print the file you have just transferred onto your PC, you must open it in an editor like notepad and must issue a print command from there, by clicking on the File menu and then selecting the Print command from there.

If you have difficulties because you are missing ftp software, have a MAC, etc. you may ask for help at the help desk at any of the sites.

A.2 Another Example of Using Your Home Computer in CISC 105

You were told to retrieve files using Netscape for Labs, etc. at the URL http://www.udel.edu/CIS/105/ — for example, lab2.c is at URL http://www.udel.edu/CIS/105/Labs/lab2.c

This is great; HOWEVER, IF you do this on your PC at home (instead of on a UD X-terminal logged into strauss), THEN the copy of lab2.c will be on your home PC's hard disk (secondary storage), WHEN YOU NEED IT TO BE ON YOUR STRAUSS ACCOUNT'S SECONDARY STORAGE. SO, here's A way to handle this (on a home PC — you don't need to do this at at UD X-terminal logged into strauss although it's ok there too):

Logged into strauss from home or from UD (see above re how to log onto strauss from home), in YOUR directory where YOU want to store a copy of, say, lab2.c, execute at the Unix prompt:

```
cp /www/htdocs/CIS/105/Labs/lab2.c .
```

The . just above in the Unix command IS PART of the command. It refers to YOUR current directory.

Note: `/www/htdocs/CIS/105/Labs/` is the UNIX directory corresponding to the URL `http://www.udel.edu/CIS/105/Labs/` — etc.

Appendix B

Useful Unix Commands

The following tables include brief descriptions of some of the Unix commands you will use most often in doing your work for this course. You should become familiar with these commands as quickly as possible - certainly by the end of your third lab.

File-related commands	
cat *filename*	display the contents of the file called *filename* on the screen
cp *filename newfilename*	copy the file called *filename* to a new file *newfilename*, keeping a replica in the file *filename*, wipes out old contents of *newfilename*
cp *filename dir_name*	copy a file *filename* into directory *dir_name*
more *filename*	display the contents of the file called *filename* on the screen, pausing when the screen is full, waiting for the user to type RETURN or ENTER to continue displaying
mv *filename newfilename*	move the file called *filename* to a new file called *newfilename*; wipes out old contents of *newfilename*, and deletes *filename* file
mv *filename dir_name*	move the file called *filename* into the directory called *dir_name*
mv $*.c$ *dir_name*	move all C files into directory *dir_name*
qpr -q *printer filename*	print the file called *filename* on the printer called *printer* Eg. qpr -q whlps lab3a.c
rm *filename*	remove (delete) the file called *filename* from the computer system; these files are really truly gone!
Directory Commands:	
cd *dir_name*	change (or move) to the directory called *dir_name*
cd ..	change (or move) back up one level in the tree
cd	change (or move) back to your home directory, from wherever you are currently located
ls	display the names of all files in the current directory
ls *dir_name*	display names of all files in directory *dir_name*
ls $*.c$	display all files ending in .c in the current directory
ls -l	uses long format for output from the `ls` command
mkdir *dir_name*	creates a new directory called *dir_name*, which is located in the current directory
pwd	displays the complete name of your current directory includes the whole pathname, that is, every directory on the path starting at the root directory of the Unix system down the tree branches to your current directory
rmdir *dir_name*	remove an entire directory from the computer system; the directory must contain no files (be empty) for this command to actually do any deletion of the directory

Misc. Commands	
cc *progname.c*	compile the C program stored in file *progname.c*
cc *progname.c* -lm	compile the C program stored in file *progname.c* where one or more functions from `math.h` are used.
cc -o *prog progname.c*	compile the C program stored in file *progname.c* and put the executable in file called *prog*
chdgrp	display all projects which you are a member of
chdgrp *number*	make project *number* your default project - takes effect the next time you logon.
command | more	the output from *command* is displayed through more one screen at a time
cc *progname.c* | more	Use this to see your error messages one screen at a time!
du	display disk usage statistics
lint *progname.c*	check *program.c* for consistency - finds errors such as missing return statements, uninitialized variables, etc.
man *command*	to display online documentation for *command*
man ls	for example, get online help for the ls command
man man	get more information on the `man` command
newgrp *number*	temporarily change to project *number*
password	change your login password - takes effect the next day
quota	display your current disk quota
script *filename.scr*	make a record of a session for printing. Note that the file name should end in .scr. (**DO NOT** use a file name like lab2a.c - as script will then completely erase your source file!

Appendix C

The Unix tcsh Shell

Here, we will discuss some nice advantages of the Unix shell, called the tcsh shell. These advantages include a feature called filename completion, and an advanced command history mechanism. We then show you how to make tcsh your default shell.

C.1 File Name Completion

The tcsh shell enables you to have file name completion. We illustrate with an example. I just made up a file in my eecis Unix account called 'aardvarksforfunandforprofit'. Without tcsh, if I want to type some command involving this file, it is painful to have to type out its long name without errors. With tcsh it is a breeze. All I have to do is type out the first few letters, hit the Tab key, and the system types out the rest for me! How many letters? you might ask. I have several files in my home directory beginning with 'aa':

```
case@polaris:~ 864 > ls aa*
aaai.98                     aaai.s96
aaai.fall96                 aaf.sessions
aaai.icml.workshop.98       aardvarksforfunandforprofit
aaai.online.search.97       aaron
```

I have two beginning with 'aar', but only one beginning with 'aard'. Hence, if I type at the Unix prompt:

```
case@polaris:~ 865 > cat aard
```

AND, THEN, HIT THE Tab key, the SYSTEM responds with:

```
case@polaris:~ 865 > cat aardvarksforfunandforprofit
```

It completes the awful spelling for me! If I, then, hit a return, I get:

```
This file's contents is slightly longer than its name.
case@polaris:~ 866 >
```

The general rule for tcsh file name completion is: after enough initial letters of a file name are typed (at the Unix command line) to distinguish this file from others (in the directory you are in), hitting a Tab will cause the whole file name to be automatically completed. It works too on DIRECTORY names!

C.2 Command History

The tcsh shell lets you do other things, one of which I'll mention now. With tcsh, if you want to reissue a command you typed several lines back, you don't have to retype it, just hit Control p (hold the Control key while hitting a p) to go back in your command line history — each hit sends you back one command. (Note: the up arrow key can be used instead of the Control p sequence.) For example, if I do:

```
case@polaris:~ 866 > ls aar*
aardvarksforfunandforprofit   aaron
case@polaris:~ 867 >
```

And THEN hit a Control p, I get:

```
case@polaris:~ 867 > ls aar*
```

A second control p (before a return) yields:

```
case@polaris:~ 867 > cat aardvarksforfunandforprofit
```

which corresponds to TWO commands back in my command history. Etc.

C.3 Activating tcsh

Ok, so how do you activate the tcsh feature?

To activate it for ONE session (before logging out) only, at your strauss Unix prompt do:

```
case@strauss:~ 99 > exec /usr/local/bin/tcsh
case@strauss:~ 1 >
```

To have it ALSO be active each subsequent time you log in do:

```
case@strauss:~ 1 > /opt/bin/chsh
Password:
New shell:/usr/local/bin/tcsh
The changes will be made within 24 hours.
case@strauss:~ 2 >
```

where the SYSTEM prompts for your password and the new shell you want. tcsh, technically, is a shell.

For more information about the tcsh shell, type man tcsh.

Appendix D

Tailoring Your Unix Prompt

To get a Unix prompt which shows your username, your machine name, YOUR CURRENT DIRECTORY, and your command history number: FIRST ACTIVATE tcsh, then type into your .cshrc file in your home directory the following four consecutive (non-blank) lines

```
set wh=‘whoami‘
alias setprompt ’set prompt="$wh@%m:%~ %h > "’
setprompt
alias cd ’cd \!*;setprompt’
```

then save this new version of .cshrc, and, lastly, at the Unix prompt execute

```
source .cshrc
```

Your new prompt should appear and persist over changes of directory and over successive logins. For example, on Professor Case's machine polaris, his prompt looks like

```
case@polaris:~/:105 180 >
```

when in directory ‘:105’ right under his home directory.

If you would like your prompt to show also the time of day for each new command line, you could use instead of the second of the four lines above, the line:

```
alias setprompt ’set prompt="$wh@%m:%~ [%t] %h > "’
```

keeping the other three lines the same.

When this prompt is used instead, in the same directory, it looks like:

```
case@polaris:~/:105 [3:45pm] 183 >
```

For a SIMPLER prompt with name of machine and directory only you could use instead of the second of the four lines above, the line:

```
alias setprompt 'set prompt = "%m:%~ > "'
```

keeping the other three lines the same.

When this last prompt is used, in the same directory, it looks like:

```
polaris:~/:105 >
```