16.070 Introduction to Computers and Programming

February 21 Recitation 3 Spring 2002

Topics:

Review of PS1/PS2

- Suggestions for future turn-in
- Debugging Hints
- Common Programming Mistakes

New Material

- Loops
- Functions
- File I/O
- Big Example

Common Turn-In Mistakes

A number of simple steps on your part can make life more pleasant for the graders (you *want* this) and will translate into faster grading overall:

1. If a problem requires more than 1 sheet of paper to answer (as most do), **please staple** those sheets together.
2. Remember to indicate **how much time** you spent working on the problem.
3. Remember to put your **email address** in the header of your code
4. Remember to put your **name** on every paper.

Debugging Hints

In the course of working on your first two problem sets, there have inevitably been moments when your programs have not worked and you had no idea why. The process of discovering and fixing malfunctioning programs is known as debugging. Here are some tips for debugging:

1. Use printf immediately after you input a value to verify that the value was input correctly.

2. Use printf frequently to check that variables are what you expect them to be at critical points in your program. Remember to output the name of the variable whose value you are checking.
   ```
   printf("MyVar is: %d\n", MyVar);
   ```

3. Work out a test case by hand and use suggestions (1) and (2) to verify it or discover where your program is diverging from your prediction.

4. Check how many times your loops run. Is it one too many times? One too few?

5. Check the values of your variables immediately before loops. Were they initialized the way you intended?

6. More debugging hints are online at: http://www.csc.uvic.ca/~gdbrown/debug/debug.html

Common Programming Mistakes

At office hours and while grading a number of mistakes have surfaced repeatedly.
Here are a few:

1. When using scanf, the '&' is left off of variable arguments. This causes a window to pop up announcing a memory fault. You'll learn about the & operator in a few weeks, but for now, remember to use it with scanf!

2. The difference between 0.0 and 0.0f is subtle. 0.0 is a double precision floating point number (double). 0.0f is a single precision floating point number (float). The following initializations illustrate the usage differences:
```
int IntegerVar   = 0;
float FloatVar   = 0.0f;
double DoubleVar = 0.0;
```

3. Many people still don't initialize variables (see above example for good initializing). They say things like "I set the value right down here" or "That value is input with scanf". It doesn't matter. All variables should be initialized, so that there is no point in the execution of your program that the value of a variable is not known. Points will be deducted for style if this is not done.

4. Indentation is essential. In the style guide (which you should have read by now), indentation rules are *required*. After an open-brace, indent. After a close-brace, unindent. Do this while you are programming and it can help you debug.

5. Several people have made the mistake of using '=' in their if statements instead of '=='. This can be a very difficult problem to spot. A good rule of thumb is to look over each of your if statements immediately after writing them and check for this problem.

6. Planning your algorithm in advance prevents confusing code and makes debugging easier. Make sure you designed your program before writing it.

Loops

Loops will be central to your programming problems for the duration of this term. You will become an expert *designer* and *implementer* of loops.

One type of loop you will become familiar with is called a **sentinel loop**. It runs until a *sentinel* variable is changed. Typically, while loops are used for sentinel checking.

For instance, the following sentinel loop will identify the first number whose square is 64.
The sentinel variable in this case is TestNumber.

```
int FoundNumber = 0;
int TestNumber = 0;

while(!FoundNumber) {
  TestNumber = TestNumber + 1;

  if (TestNumber*TestNumber==64) {
    FoundNumber = 1;
  } /* end if sqrt 64 is found */

} /* end while to find square rt of 64 */

printf("sqrt 64 = %d\n", TestNumber);
```

The while loop is also used frequently for **input protection**. A program with looping input protection traps the user input in a loop and continues until the user enters a valid input.

For instance, the following program uses input protection to continue receiving integers until the user gives a positive value.

```
int Input = 0;
while (Input<=0) {
  printf("Input a positive integer: ");
  scanf("%d", &Input);
} /* end while */
printf("You input the positive integer: %d\n\n", Input);
```

Another common loop is the **counting loop**. The counting loop will iterate over a prescribed range. Typically, for loops are used for counting.

The following counting loop adds the squares of all natural numbers less than 20.

```
int Counter = 0;
int Sum = 0;
…
Sum = 0;
for (Counter=1; Counter<20; Counter++) {
  Sum = Sum + (Counter*Counter);
} /* end for to add squares of nat. num < 20 */

printf("The sum of the first 19 squares is %d\n\n", Sum);
```

One of the most common mistakes encountered in for loops occurs in the termination condition. For instance, what is the value of Counter when the loop terminates? What is the last value of Counter*Counter added to Sum? Be very careful when deciding whether to use "<=" or "<".

Finally, note the seemingly gratuitous use of variable assignment in the preceding example. As stated earlier, all variables should be assigned values at declaration. It is also *highly recommended* that you assign intended values to variables immediately before their use in a loop. Not doing this relies on the implicit condition that your variable is unchanged between its declaration and its use in the loop. This works for small programs, but will be completely unmanageable in larger programs. Also, setting your variables at the start of a loop makes debugging simpler, because you do not need to go scan through your program to find when the variable was last set.

Functions

Functions were introduced in yesterday's lecture. Generally, functions are used in one or both of two cases: (1) similar code is used multiple times or (2) consolidating code can make a program more readable/testable. A general rule of thumb for this class is that you cannot have too many functions. Each step in your decomposition outline should probably be a separate function. If possible, write functions that are general enough to be reused in other programs. This will reduce your future coding burden. Also, using previously tested functions tends to make code more reliable.

Every function will have known inputs and known outputs. These should be specified in the function's comment header (the style guide has many instructions for function style). A function can be tested by observing its outputs for given inputs. A test harness is a program that is designed to test a function or set of functions.

The following is an example of a well-written, well-documented, reusable function.

```
/*********************************/
/* Function: Distance            */
/*   This function computes the  */
/*   absolute distance between   */
/*   its two integer inputs      */
/*                               */
/* Inputs: int Num1, int Num2    */
/*         no preconditions      */
/* Output: abs(Num1-Num2)        */
/*********************************/

int Distance(int Num1, int Num2) {

  int Length = 0; /* holds the temporary length value */

  Length = Num1 - Num2;

  if (Length<0) {
    Length = Length * -1;
  } /* end if Length negative */

  return Length;
} /* end int Distance */
/************************************************/
```

A test harness for the function ”Distance” might be:

```
/*************************************/
/* Test Harness for Distance         */
/* by Louis Breger (lbreger@mit.edu) */
/*                                   */
/* This program will test the function */
/* length by providing it with inputs  */
/* -5 through 5 combined with 0        */
/*************************************/

/* Preprocessor Directives */
#include <stdio.h>

/* Function Prototypes */
int Distance(int Num1, int Num2);

/**** MAIN ****/
int main(void) {

  int Counter = 0;          /* counter variable */
  int CurrentD = 0;  /* holding variable for distances */

  for (Counter = -5; Counter<6; Counter++) {
    CurrentD = Distance(Counter, 0);
    printf("The distance between %d and 0 is %d\n", Counter, CurrentD);
  } /* end for to test Distance */

  printf("\n\n");

} /* end main */
```

File I/O (Input/Output)

In order to use file I/O you must declare every input and output file with the following syntax:
```
FILE * file_name;
```

This declares the internal *file_name* for the file you want to operate on, sometimes called a "file handle". Next you must point this internal file name to an external file using the *fopen* command. The format of the *fopen* command is as follows:
```
file_name = fopen("external_file_name", Openmode);
```

Where *Openmode* is one of the following:

Open Mode	Resulting Permissions
"r"	read access
"w"	write access
"a"	append data to the end of an existing file
"r+"	opens a file for input & output, if it exists already, the contents are not destroyed
"w+"	destroys the file if it exists and creates a new one for update
"a+"	opens a file for update so that writing is done at the end of the file

The "w" and "w+" modes are destructive! The "a" and "a+" modes are not destructive. They append data to already existing files. The difference is that you can read from a file that was opened with "a+" but not one that was opened with "a".
If the *fopen* command fails for some reason the function will return a NULL pointer value. This can be useful when determining whether a file has opened successfully or not. After opening and operating on a file it must be closed.
This is done with the *fclose* command:
```
fclose(file_name);
```

Another very useful function is the *feof* function. It returns a true (1) if the most recent input operation on the file returned an end of file character, and returns false (0) otherwise. There is also a constant EOF that is defined to represent the end of file character. This can also be used in various ways to test for the end of a file.

Writing Data to Files

The *fprintf* function is used to write data to a file. It is very similar to the *printf* function except it requires a file name to write to.
```
fprintf(file_name, format_specifiers, variables);
```

There are two other functions that can be used to write to files. The *fputc* function writes a character to a file and the *fputs* function writes a string to a file.
```
fputc(CharacterExpression, file_name);
fputs(StringExpression, file_name);
```

Reading Data from Files

The functions used to read data from files are similar to those seen earlier. The three functions are *fscan*f, *fget*c, and *fget*s.
```
fscanf(file_name, format_specifiers, variables);
fgetc(file_name);
fgets(StringExpression, Size, file_name);
```

Since *fgetc* has no variable specification it is usually used in an expression in the form of
```
CharacterVariable = fgetc(file_name);
```

Input from Preformatted Data Files

The data can be contained in fixed width fields, it is possible to read it by simply using the proper width declaration in the format specifiers of the *fscanf* function. The data can also be separated by a specific character, such as a comma in a comma separated value (CSV) file. It is possible to store the separation character in a dummy variable or include the character between the specifiers in the function call.

File I/O Examples

Assuming a file DataFile has been opened for output, the following table shows output statements and their corresponding results.

`fprintf(DataFile, "%d\n", 20)`	`20↵`
`fprintf(DataFile, "%d%d\n", 20, 40)`	`2040↵`
`fprintf(DataFile, "%f\n", 16.25)`	`16.250000↵`
`fprintf(DataFile, "%15e\n", 16.25)`	`1.625000e+01↵`
`fprintf(DataFile, "%c%c\n", 'a', 'b')`	`ab↵`

`fputc('a', DataFile);`
`fputc('b', DataFile);`
`fputc('\n', DataFile);`
	`ab↵`

`fputs("16.070 is great.\n", DataFile)`	`16.070 is great.↵`

The following program opens the file R3_InputExample.txt and prints its contents to the screen.

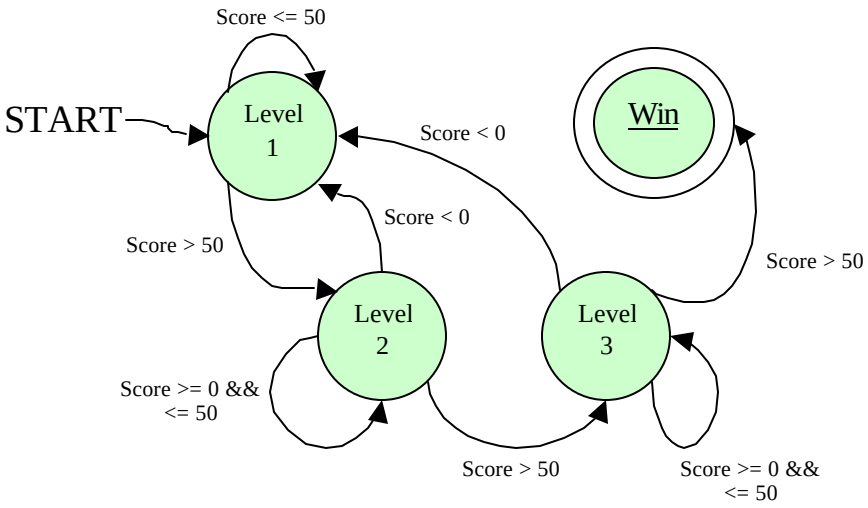
```
#include <stdio.h>
#define InFileName "R3_InputExample.txt"

int main(void) {
  FILE *InputFile;  /* declare a file variable */
  int InputVar = 0;

  InputFile = fopen(InFileName, "r");  /* open a file for reading */

  if (InputFile != NULL) {             /* check if file opened */
    while ( !feof(InputFile) ) {         /* read ints until end of file */
      fscanf(InputFile, "%d", &InputVar);  /* read int from the file */
      printf("%d\n", InputVar);
    } /* end while not at end of file */
  } /* end if no input error */
  else {
    printf("Input file could not be opened.\n");
  } /* end if input error */
  fclose(InputFile);  /* close file */
} /* end main */
```

This program accepts user input from the keyboard and prints it to a file.

```
#include <stdio.h>
#define OutFileName "R3_OutputExample.txt"

int main(void) {
  FILE *OutputFile;  /* declare a file variable */
  int InputVar = 0;  /* holds file input */

  OutputFile = fopen(OutFileName, "w+");  /* open a file for output */

  if (OutputFile != NULL) {  /* check if file opened */
    printf("Type numbers to be written to a file. Enter 0 to stop.\n");
    scanf("%d", &InputVar);  /* read an int from the file */

    while ( InputVar != 0 ) {  /* print & read ints until 0 is read */
      fprintf(OutputFile, "%d\n", InputVar);
      scanf("%d", &InputVar);
    } /* end while input not 0*/
  } /* end if no input error */
  else {
    printf("Output file could not be opened.\n");
  } /* end if input error */

  fclose(OutputFile);  /* close file */
} /* end main */
```

Cumulative Recitation Example: State Transition Diagrams (aka FSAs)

The following program implements the state transition diagram shown below. The diagram describes a competition, whereby a player begins in level 1 and proceeds on to the next level by getting a score greater than 50, stays in the current level by getting a score between 0 and 50 (inclusive), and goes back to the first level by getting a negative score. The competition ends when the player progresses beyond level 3. The player's scores will be read in from a data file. The programming implementation shown here is by no means the only way to code a state chart. It's just an example of a simple, highly modular approach that facilitates easy modification.




```
/*******************************/
/* This program determines the  */
/*   smallest angle between two */
/*   headings.                  */
/* by Louis Breger, Feb 2002    */
/* For 16.070 Recitation 2      */
/*******************************/

/* Preprocessor Directives */
#include <stdio.h>

#define InFile "R3_StateTransitionInput.txt"

#define LEVEL1 1
#define LEVEL2 2
#define LEVEL3 3
#define WIN    4

/* Function Prototypes */
int State_Level1(int Score);
int State_Level2(int Score);
int State_Level3(int Score);

/*************************************************/
int main(void)  {

  /* Variable Declarations */
  int State = LEVEL1; /* tracks current state   */
                      /* expected range: [1,4]  */

  int Score = 0;      /* variable holds player's score */
  FILE * ScoreData;   /* File holds the player's scores */

  ScoreData = fopen(InFile, "r"); /* open score file */
```

```
  if (ScoreData) {

    while (!feof(ScoreData) && State!=WIN) {

      fscanf(ScoreData, "%d", &Score);

      switch (State) {
        case LEVEL1:
          State = State_Level1(Score);
          break;
        case LEVEL2:
          State = State_Level2(Score);
          break;
        case LEVEL3:
          State = State_Level3(Score);
          break;
        default: printf("error: unknown state\n");
      } /* end switch to implement state */
    } /* end while */

    /* output results */
    if (State==WIN) {
      printf("You won.\n\n");
    } /* end if won */
    else {
      printf("You reached Level %d\n\n", State);
    } /* end else did not win */

    fclose(ScoreData);  /* close score file */

  } /* end if file opened correctly */

  else {
    printf("Error opening file.\n");
  } /* end else error opening file */

  return 0;
} /* end main */
/*****************************************************/


/***********************************/
/* Function: State_Level1          */
/*   This function will implement  */
/*   the FSA node Level 1          */
/*                                 */
/* Inputs: an integer score        */
/* Output: integer indicating the  */
/*         next state              */
/***********************************/
int State_Level1(int Score) {
  int NextState = -1;

  printf("Entering State 1 with score %d\n", Score);
  if (Score <= 50) {
    NextState = LEVEL1;
  } /* end if score <= 50 */
  else if (Score > 50) {
    NextState = LEVEL2;
  } /* end else if score >50 */
  else {
    printf("State_Level1: error, invalid score\n");
  } /* end else error */

  return NextState;
} /* end State_Level1 */
/*****************************************************/
```

```
/**********************************/
/* Function: State_Level2           */
/* Comment truncated to save trees  */
/**********************************/
int State_Level2(int Score) {
  int NextState = -1;

  printf("Entering State 2 with score %d\n", Score);
  if (Score < 0) {
    NextState = LEVEL1;
  } /* end if score < 0 */
  else if ((Score >= 0) && (Score <= 50)) {
    NextState = LEVEL2;
  } /* end else if score >=0 and <= 50 */
  else if (Score > 50) {
    NextState = LEVEL3;
  } /* end else if score >50 */
  else {
    printf("State_Level2: error, invalid score\n");
  } /* end else error */

  return NextState;
} /* end State_Level2 */
/*************************************************/


/**********************************/
/* Function: State_Level3           */
/* Comment truncated to save trees  */
/**********************************/
int State_Level3(int Score) {
  int NextState = -1;

  printf("Entering State 3 with score %d\n", Score);
  if (Score < 0) {
    NextState = LEVEL1;
  } /* end if score < 0 */
  else if ((Score >= 0) && (Score <= 50)) {
    NextState = LEVEL3;
  } /* end else if score >=0 and <= 50 */
  else if (Score > 50) {
    NextState = WIN;
  } /* end else if score >50 */
  else {
    printf("State_Level3: error, invalid score\n");
  } /* end else error */

  return NextState;
} /* end State_Level3 */
/*************************************************/
```

Homework 3 Design Problem Notes

In the design problem (ps3p5), you are asked to design a program that implements a controller specified by a finite state automata (FSA). Your answer to this problem will almost certainly make use of the concepts contained in this recitation. When designing the program, think carefully about all of the conditions contained in the FSA. When you code the program, you may wish to test your input routines separately to verify that your file I/O is working properly. Consider testing the functions in your program individually to verify anticipated performance. Also, be sure you test your solution on a variety of complicated test cases that use many state transitions… we will!