16.070 Introduction to Computers and Programming

15 February Recitation 2 Spring 2001

Topics

Proper Homework Format

- Turn in requirements for software design problems
- Special formatting guidelines for homework
- Programming guide

Steps for Programming Algorithms

- Concept definition & problem specification
- Requirements & analysis
- Design
- Programming
- Testing
- Maintenance

Turn in requirements for software design problems

When doing the homework you should follow the Spiral Model of the Software Design Process. This model calls for you to perform each of the steps at least twice. We will require that you only turn in the final version of your work for each stage. The material that must be turned in for each stage is listed in the example that we will step through today. The format of each item can be found in the next section.

Each Problem Set will contain Turn In requirements associated with each of its problems. These requirements should be adhered to. In general however, you will be required to turn in at least your program code (hardcopy) and a screen dump of its output. Some problems might require you to draw up a flow chart or write pseudo code only. Always use some common sense too, turning in information that clarifies your answers whenever you feel that its necessary.

Step 1: Concept

This portion of the process is done for you in the form of the problem statement.

Example

Design a program that reads the number of letter grades A, B, C, D and F for a student; computes and prints the student's GPA. When computing the GPA assume that an A = 5, B = 4, C = 3, D = 2 and F = 0. Below is a sample of what your program should look like when it is run.

```
Enter the number of As:3
Enter the number of Bs:2
Enter the number of Cs:2
Enter the number of Ds:0
Enter the number of Fs:0

Your GPA is 4.142857

Press any key to continue
```

Step 2: Requirement Stage

1. *Detailed Requirements;* list that contains each individual software requirement
2. *Analysis of Requirement*s; list all of the inputs, outputs, constraints, and formulas that are required for the software

Example

Detailed Requirements

- Prompt the user to enter the number of each letter grade
- Track number of each letter grade
- Calculate the GPA
- Display the GPA

Analysis

- Inputs: number of each letter grades (5)
- Outputs: GPA
- Formulas: GPA = (#a * 5 + #b * 4 + #c * 3 + #d * 2) / (#a + #b + #c + #d + #f)

Step 3: Design

A algorithm that needs to be programmed can be broken up into conceptually different modules that are interconnected. Together these modules make up the whole of the algorithm. Each module may eventually be coded (programmed) as single or multiple functions. Our example requires only one function to implement each module.

1. *Modules;* list that contains the names of all the modules you intend to use and the problems that those modules solve
2. *Module Algorithm*s; write out the algorithms that you will use to solve the problem in each of the modules (Pseudocode, flow chart/state diagrams)
3. *Test Cases;* list that contains all of the test cases you intend to run on the finished software and the reason for choosing each

Module Guidelines

- Should be described using 1-2 short English sentences
- Make names of modules reflect their purpose

Representation of an Algorithm

You should write out Pseudo-Code for each of the algorithms you intend to use or include a Flow Chart or a State Transition Diagram for each. Pseudo-code (required for our example) should contain a description of each of the algorithm steps written out in English. The vocabulary that you use to write your pseudo-code should be limited. It may contain words such as:

```
begin, end, if, then, while, print, read, etc…
```

Each pseudo-code statement should be written on a separate line and if a statement takes more than one line, the continuation lines should be indented. Different blocks of pseudo-code should be indented from one and other.

When drawing flow charts or state transition diagrams you should follow the specifications contained in the Real-Time Systems Book.

Example

Modules

- **get_number** prompt user for a number reads in value
- **calculate_gpa** calculate the GPA
- **display_results** displays the GPA
- **main** module linking other modules together to evaluate algorithm

Module Algorithms

- *get_number(Grade_Chr)*

  print "Enter the number of *GradeChr*s:"
  read number of *GradeChr*s
  return number of *GradeChr*s

- *calculate_gpa*

  gpa = (As*5 + Bs*4 + Cs*3 + Ds*2) / (As + Bs+ Cs + Ds + Fs)
  return gpa

- *display_results*

  print "Your GPA is gpa"

- *main*

  get_number of A's
  get_number of B's
  get_number of C's
  get_number of D's
  get_number of F's

  calculate_gpa

  display_results

Test Cases

Generate a number of grades and determine the GPA by hand. Then check whether the program concurs.

Step 4: Programming

1. Source code printout
2. Screen dump of sample run
3. Electronic submission of source code, when required

Example

```
/* This program determines your GPA from the number
   of specific letter grades that you enter */
/* Written by btk Dec 2000, modified by tj on 10 Feb. 2001 */
/* For 16.070 Recitation 2 */

#include <stdio.h>

/* Function Prototypes */
/* See function definitions below for details */
int get_number(char Letter_Grade);
float calculate_GPA(int As, int Bs, int Cs, int Ds, int Fs);
void display_results(float GPA);

/* Main Function  */
int main(void)
{
       /* Variable Declarations */
       int As, Bs, Cs, Ds, Fs;    /* Number of letter grades */
```

```
	float GPA;		/* Grade Point Average */

	/* Prompt user for # of letter grades and store them */
	As = get_number('A');
	Bs = get_number('B');
	Cs = get_number('C');
	Ds = get_number('D');
	Fs = get_number('F');

	/* Calculate the GPA */
	GPA = calculate_GPA(As, Bs, Cs, Ds, Fs);

	/* Display the results */
	display_results(GPA);

	return 0;
}	/* end main */


/*-----------------------------------------------------------
// Prompts the user to enter the number of Grades
//-----------------------------------------------------------*/

int get_number(char Letter_Grade)
{
	/* Declare local Variables */
	int Letter_GradeS;

	/* Prompt User to enter number of letter grades */
	printf("Enter the number of %cs:", Letter_Grade);
	scanf("%d", &Letter_GradeS);

	/* Return the number of Letter_Grade s */
	return Letter_GradeS;
}


/*-----------------------------------------------------------
// Calculate the GPA given number of each letter grade
//-----------------------------------------------------------*/

float calculate_GPA(int As, int Bs, int Cs, int Ds, int Fs)
{
	/* Local Variable Declaration */
	float GPA;

	/* Calculate the GPA */
	GPA = ( As * 5.0 + Bs * 4.0 + Cs * 3.0 + Ds * 2.0) / (As + Bs + Cs + Ds + Fs );

	/* Return value of GPA */
	return GPA;
}


/*-----------------------------------------------------------
// Display the GPA
//-----------------------------------------------------------*/

void display_results(float GPA)
{
	/* Print GPA to screen*/
	printf("\n");
	printf("Your GPA is %f\n\n", GPA);
}
```

Step 5: Test Phase

Screendumps of the test cases and their outputs.

Example

```
Enter the number of As:3
Enter the number of Bs:2
Enter the number of Cs:2
Enter the number of Ds:0
Enter the number of Fs:0

Your GPA is 4.142857

Press any key to continue
```

Step 6: Maintenance Phase

For this example: This is not a large program, but a correct coding style, keeping readability in mind is required. Remember to add comments and indent your code. This enables a different programmer to follow your code and make additions or changes. (See extract of the programming guide below.) For our purposes, you don't need to explicitly include a discussion about maintenance.

Programming Guide

Readability of Code

- A comment should come before each block of code that describes its function
- Different blocks of code should start in different columns
- Insert blank lines between different blocks of code
- One statement per line is allowed (a ; should be followed by an enter)
- Meaningful variable names should be used
- Avoid running statements over multiple lines
- Use whitespace in expressions (before and after operators)

Function Guidelines

- Use global prototypes of functions
- Use a comment separator before each function and a comment that describes the purpose and arguments of the function
- Functions should appear in programs in the order in which they are used
- Optimize module size (2-50 lines of code)
- Restrict number of functions called by a function (rule of thumb ~7)
- Assign descriptive names to functions that reflect their purpose
- Clarify and Identify all I/O in comments before functions
- Use parameters to pass data, not global variables
- Design your functions to be used in other programs
- Use functions that have been defined and tested