```
# and the thing(s) to the right of the colon are what it depends on.  The second
# line is the action to create the target.  If the things it depends on
# haven't changed since the target was last created, no need to do the action.
# Note:  The tab spacing in the second line is important!  You can't just use
# individual spaces.

# "make myprog" or "make" links two object codes to create the executable
myprog:  main.o rad2volume.o
	gcc main.o rad2volume.o -o myprog

# "make main.o" produces main.o object code; depends on main.c and rad2volume.h
main.o:  main.c rad2volume.h
	gcc -c main.c -o main.o

# "make rad2volume.o" produces rad2volume.o; depends on one .c and one h file
rad2volume.o:  rad2volume.c rad2volume.h
	gcc -c rad2volume -o rad2volume.o

# "make clean" throws away any object files to ensure make from scratch
clean:
	rm *.o
```

With this `Makefile` in the same directory as your other files, you should be able to type the command `make [target]`,[26] where `[target]` is `myprog`, `main.o`, `rad2volume.o`, or `clean`. If the `target` depends on other files, `make` will make sure those are up to date first, and if not, it will call the commands needed to create them. For example, `make myprog` triggers a check of `main.o`, which triggers a check of `main.c` and `rad2volume.h`. If either of those have changed since the last time `main.o` was made, then `main.c` is compiled and assembled to create a new `main.o` before the linking step.

The command `make` with no target specified will make the first target (which is `myprog` in this case).

Ensure that your `Makefile` is saved without any extensions (e.g., `.txt`) and that the commands are preceded by a tab (not spaces).

There are many more sophisticated uses of `Makefile`s which you can learn about from other sources.

A.5 Exercises

1. Install C, create the `HelloWorld.c` program, and compile and run it.
2. Explain what a pointer variable is, and how it is different from a non-pointer variable.
3. Explain the difference between interpreted and compiled code.

[26] In some C installations `make` is named differently, like `nmake` for Visual Studio or `mingw32-make`. If you can find no version of `make`, you may not have selected the `make` tools installation option when you performed the C installation.

4. Write the following hexadecimal (base-16) numbers in eight-digit binary (base-2) and three-digit decimal (base-10). Also, for each of the eight-digit binary representations, give the value of the most significant bit. (a) 0x1E. (b) 0x32. (c) 0xFE. (d) 0xC4.
5. What is 333_{10} in binary and 1011110111_2 in hexadecimal? What is the maximum value, in decimal, that a 12-bit number can hold?
6. Assume that each byte of memory can be addressed by a 16-bit address, and every 16-bit address has a corresponding byte in memory. How many total bits of memory do you have?
7. (Consult the ASCII table.) Let `ch` be of type `char`. (a) The assignment `ch = 'k'` can be written equivalently using a number on the right side. What is that number? (b) The number for `'5'`? (c) For `'='`? (d) For `'?'`?
8. What is the range of values for an `unsigned char`, `short`, and `double` data type?
9. How many bits are used to store a `char`, `short`, `int`, `float`, and `double`?
10. Explain the difference between `unsigned` and `signed` integers.
11. (a) For integer math, give the pros and cons of using `char`s vs. `int`s. (b) For floating point math, give the pros and cons of using `float`s vs. `double`s. (c) For integer math, give the pros and cons of using `char`s vs. `float`s.
12. The following `signed short int`s, written in decimal, are stored in two bytes of memory using the two's complement representation. For each, give the four-digit hexadecimal representation of those two bytes. (a) 13. (b) 27. (c) −10. (d) −17.
13. The smallest positive integer that cannot be exactly represented by a four-byte IEEE 754 `float` is $2^{24} + 1$, or 16,777,217. Explain why.
14. Give the four bytes, in hex, that represent the following decimal values: (a) 20 as an `unsigned int`. (b) −20 as a two's complement `signed int`. (c) 1.5 as an IEEE 754 `float`. (d) 0 as an IEEE 754 `float`.

    To verify your answers, use the program `typereps.c` below. This program allows the user to enter four bytes as eight hex characters, then prints the value of those four bytes when they are interpreted as an `unsigned int`, a two's complement `signed int`, an IEEE 754 `float`, or four consecutive `char`s. To do this, the program creates a new data type `four_types_t` consisting of four bytes, or 32 bits. These same 32 bits are interpreted as either an `unsigned int`, `int`, `float`, or four `char`s depending on whether we reference the bits using the fields `u`, `i`, `f`, or `char0`–`char3` in the `union`.

```
#include <stdio.h>

typedef union {    // a new data type consisting of four bytes
  unsigned int u; // the 32 bits interpreted as an unsigned int
  int i;          // the same 32 bits interpreted as a two's complement int
  float f;        // the same 32 bits interpreted as an IEEE 754 single prec float
  struct {
    char char0:8; // bits  0 -  7 interpreted as char, called char0
    char char1:8; // bits  8 - 15 interpreted as char, called char1
    char char2:8; // bits 16 - 23 interpreted as char, called char2
```

```
      char char3:8; // bits 24 - 31 interpreted as char, called char3
    };
  } four_types_t;  // the new type is called four_types_t

  int main(void) {
    four_types_t val;

    while (1) {     // exit the infinite loop using ctrl-c or similar
      printf("Enter four bytes as eight hex characters 0-f, e.g., abcd0123:  ");
      scanf("%x",&val.u);
      printf("\nThe 32 bits in hex:                           %x\n",val.u);
      printf("The 32 bits as an unsigned int, in decimal:  %u\n",val.u);
      printf("The 32 bits as a signed int, in decimal:     %d\n",val.i);
      printf("The 32 bits as a float:                      %.20f\n",val.f);
      printf("The 32 bits as 4 chars:                      %c %c %c %c\n\n",
             val.char3, val.char2, val.char1, val.char0);
    }
    return 0;
  }
```

Below is a sample output. Note that only the ASCII values 32-126 have a visible printed representation, so the printouts as `char`s are meaningless in the first two examples.

```
Enter four bytes as eight hex characters 0-f, e.g., abcd0123: c0000000

The 32 bits in hex:                          c0000000
The 32 bits as an unsigned int, in decimal:  3221225472
The 32 bits as a signed int, in decimal:     -1073741824
The 32 bits as a float:                      -2.00000000000000000000
The 32 bits as 4 chars:                      ?

Enter four bytes as eight hex characters 0-f, e.g., abcd0123: ff800000

The 32 bits in hex:                          ff800000
The 32 bits as an unsigned int, in decimal:  4286578688
The 32 bits as a signed int, in decimal:     -8388608
The 32 bits as a float:                      -inf
The 32 bits as 4 chars:                      ? ?

Enter four bytes as eight hex characters 0-f, e.g., abcd0123: 48494a4b

The 32 bits in hex:                          48494a4b
The 32 bits as an unsigned int, in decimal:  1212762699
The 32 bits as a signed int, in decimal:     1212762699
The 32 bits as a float:                      206121.17187500000000000000
The 32 bits as 4 chars:                      H I J K
```

You can easily modify the code to allow the user to enter the four bytes as a `float` or `int` to examine their hex representations.

15. Write a program that prints out the sign, exponent, and significand bits of the IEEE 754 representation of a `float` entered by the user.
16. Technically the data type of a pointer to a `double` is "pointer to type `double`." Of the common integer and floating point data types discussed in this chapter, which is the most similar to this pointer type? Assume pointers occupy eight bytes.

17. To keep things simple, let us assume we have a microcontroller with only $2^8 = 256$ bytes of RAM, so each address is given by a single byte. Now consider the following code defining four global variables:

```
unsigned int i, j, *kp, *np;
```

Let us assume that the linker places `i` in addresses 0xB0..0xB3, `j` in 0xB4..0xB7, `kp` in 0xB8, and `np` in 0xB9. The code continues as follows:

```
              // (a) the initial conditions, all memory contents unknown
kp = &i;      // (b)
j = *kp;      // (c)
i = 0xAE;     // (d)
np = kp;      // (e)
*np = 0x12;   // (f)
j = *kp;      // (g)
```

For each of the comments (a)-(g) above, give the contents (in hexadecimal) at the address ranges 0xB0..0xB3 (the `unsigned int i`), 0xB4..0xB7 (the `unsigned int j`), 0xB8 (the pointer `kp`), and 0xB9 (the pointer `np`), at that point in the program, after executing the line containing the comment. The contents of all memory addresses are initially unknown or random, so your answer to (a) is "unknown" for all memory locations. If it matters, assume little-endian representation.

18. Invoking the `gcc` compiler with a command like `gcc myprog.c -o myprog` actually initiates four steps. What are the four steps called, and what is the output of each step?
19. What is `main`'s return type, and what is the meaning of its return value?
20. Give the `printf` statement that will print out a `double d` with eight digits to the right of the decimal point and four spaces to the left.
21. Consider three `unsigned char`s, `i`, `j`, and `k`, with values 60, 80, and 200, respectively. Let `sum` also be an `unsigned char`. For each of the following, give the value of `sum` after performing the addition. (a) `sum = i+j;` (b) `sum = i+k;` (c) `sum = j+k;`
22. For the variables defined as

```
int a=2, b=3, c;
float d=1.0, e=3.5, f;
```

give the values of the following expressions. (a) `f = a/b;` (b) `f = ((float) a)/b;` (c) `f = (float) (a/b);` (d) `c = e/d;` (e) `c = (int) (e/d);` (f) `f = ((int) e)/d;`

23. In each snippet of code in (a)-(d), there is an arithmetic error in the final assignment of `ans`. What is the final value of `ans` in each case?

a.
```
char c = 17;
float ans = (1 / 2) * c;
```
b.
```
unsigned int ans = -4294967295;
```
c.
```
double d = pow(2, 16);
short ans = (short) d;
```
d.
```
double ans = ((double) -15 * 7) / (16 / 17) + 2.0;
```

24. Truncation is not always bad. Say you wanted to store a list of percentages rounded down to the nearest percent, but you were tight on memory and cleverly used an array of `char`s to store the values. For example, pretend you already had the following snippet of code:

```
char percent(int a, int b) {
  // assume a <= b
  char c;
  c = ???;
  return c;
}
```

You cannot simply write `c = a / b`. If $\frac{a}{b} = 0.77426$ or $\frac{a}{b} = 0.778$, then the correct return value is `c` = 77. Finish the function definition by writing a one-line statement to replace `c = ???`.
25. Explain why global variables work against modularity.
26. What are the seven sections of a typical C program?
27. You have written a large program with many functions. Your program compiles without errors, but when you run the program with input for which you know the correct output, you discover that your program returns the wrong result. What do you do next? Describe your systematic strategy for debugging.
28. Erase all the comments in `invest.c`, recompile, and run the program to make sure it still functions correctly. You should be able to recognize what is a comment and what is not. Turn in your modified `invest.c` code.
29. The following problems refer to the program `invest.c`. For all problems, you should modify the original code (or the code without comments from the previous problem) and run it to make sure you get the expected behavior. For each problem, turn in the modified portion of the code only.
    a. *Using* `if`, `break` *and* `exit`. Include the header file `stdlib.h` so we have access to the `exit` function (Section A.4.14). Change the `while` loop in `main` to be an infinite loop by inserting an expression `<expr>` in `while(<expr>)` that always evaluates to 1 (TRUE). (What is the simplest expression that evaluates to 1?) Now the first command inside the while loop gets the user's input. `if` the input is not valid, `exit` the program; otherwise continue. Next, change the `exit` command to a `break` command, and see the different behavior.
    b. *Accessing fields of a* `struct`. Alter `main` and `getUserInput` to set `inv.invarray[0]` in `getUserInput`, not `main`.
    c. *Using* `printf`. In `main`, before `sendOutput`, echo the user's input to the screen. For example, the program could print out `You entered 100.00, 1.05, and 5.`
    d. *Altering a string*. After the `sprintf` command of `sendOutput`, try setting an element of `outstring` to 0 before the `printf` command. For example, try setting the third element of `outstring` to 0. What happens to the output when you run the program? Now try setting it to `'0'` instead and see the behavior.

   e. *Relational operators*. In `calculateGrowth`, eliminate the use of `<=` in favor of an equivalent expression that uses `!=`.
   f. *Math*. In `calculateGrowth`, replace `i=i+1` with an equivalent statement using `+=`.
   g. *Data types*. Change the fields `inv0`, `growth`, and `invarray[]` to be `float` instead of `double` in the definition of the `Investment` data type. Make sure you make the correct changes everywhere else in the program.
   h. *Pointers*. Change `sendOutput` so that the second argument is of type `int *`, i.e., a pointer to an integer, instead of an integer. Make sure you make the correct changes everywhere else in the program.
   i. *Conditional statements*. Use an `else` statement in `getUserInput` to print `Input is valid` if the input is valid.
   j. *Loops*. Change the `for` loop in `sendOutput` to an equivalent `while` loop.
   k. *Logical operators*. Change the assignment of `valid` to an equivalent statement using `||` and `!`, and no `&&`.

30. Consider this array definition and initialization:

```
int x[4] = {4, 3, 2, 1};
```

For each of the following, give the value or write "error/unknown" if the compiler will generate an error or the value is unknown. (a) `x[1]` (b) `*x` (c) `*(x+2)` (d) `(*x)+2` (e) `*x[3]` (f) `x[4]` (g) `*(&(x[1]) + 1)`

31. For the (strange) code below, what is the final value of `i`? Explain why.

```
int i,k=6;
i = 3*(5>1) + (k=2) + (k==6);
```

32. As the code below is executed, give the value of `c` in hex at the seven break points indicated, (a)-(g).

```
unsigned char a=0x0D, b=0x03, c;
c = ~a;      // (a)
c = a & b;   // (b)
c = a | b;   // (c)
c = a ^ b;   // (d)
c = a >> 3;  // (e)
c = a << 3;  // (f)
c &= b;      // (g)
```

33. In your C installation, or by searching on the web, find a listing of the header file `stdio.h`. Find the function prototype for one function provided by the library, but not mentioned in this appendix, and describe what that function does.
34. Write a program to generate the ASCII table for values 33 to 127. The output should be two columns: the left side with the number and the right side with the corresponding character. Turn in your code and the output of the program.
35. We will write a simple *bubble sort* program to sort a string of text in ascending order according to the ASCII table values of the characters. A bubble sort works as follows.

Given an array of n elements with indexes 0 to $n-1$, we start by comparing elements 0 and 1. If element 0 is greater than element 1, we swap them. If not, leave them where they are. Then we move on to elements 1 and 2 and do the same thing, etc., until finally we compare elements $n-2$ and $n-1$. After this, the largest value in the array has "bubbled" to the last position. We now go back and do the whole thing again, but this time only comparing elements 0 up to $n-2$. The next time, elements 0 to $n-3$, etc., until the last time through we only compare elements 0 and 1.

Although this simple program `bubble.c` could be written in one function (`main`), we are going to break it into some helper functions to get used to using them. The function `getString` will get the input from the user; the function `printResult` will print the sorted result; the function `greaterThan` will check if one element is greater than another; and the function `swap` will swap two elements in the array. With these choices, we start with an outline of the program that looks like this.

```
#include <stdio.h>
#include <string.h>
#define MAXLENGTH 100       // max length of string input

void getString(char *str);  // helper prototypes
void printResult(char *str);
int greaterThan(char ch1, char ch2);
void swap(char *str, int index1, int index2);

int main(void) {
  int len;                  // length of the entered string
  char str[MAXLENGTH];      // input should be no longer than MAXLENGTH
  // here, any other variables you need

  getString(str);
  len = strlen(str);        // get length of the string, from string.h
  // put nested loops here to put the string in sorted order
  printResult(str);
  return(0);
}

// helper functions go here
```

Here's an example of the program running. Everything after the first colon is entered by the user. Blank spaces are written using an underscore character, since `scanf` assumes that the string ends at the first whitespace.

```
Enter the string you would like to sort:  This_is_a_cool_program!
Here is the sorted string:  !T____aacghiilmoooprrss
```

Complete the following steps in order. Do not move to the next step until the current step is successful.

a. Write the helper function `getString` to ask the user for a string and place it in the array passed to `getString`. You can use `scanf` to read in the string. Write a simple call in `main` to verify that `getString` works as you expect before moving on.

b. Write the helper function `printResult` and verify that it works correctly.
c. Write the helper function `greaterThan` and verify that it works correctly.
d. Write the helper function `swap` and verify that it works correctly.
e. Now define the other variables you need in `main` and write the nested loops to perform the sort. Verify that the whole program works as it should.

Turn in your final documented code and an example of the output of the program.

36. A more useful sorting program would take a series of names (e.g., `Doe_John`) and scores associated with them (e.g., 98) and then list the names and scores in two columns in descending order. Modify your bubble sort program to do this. The user enters a name string and a number at each prompt. The user indicates that there are no more names by entering `0 0`.

    Your program should define a constant `MAXRECORDS` which contains the maximum number of records allowable. You should define an array, `MAXRECORDS` long, of `struct` variables, where each `struct` has two fields: the name string and the score. Write your program modularly so that there is at least a `sort` function and a `readInput` function of type `int` that returns the number of records entered.

    Turn in your code and example output.

37. Modify the previous program to read the data in from a file using `fscanf` and write the results out to another file using `fprintf`. Turn in your code and example output.

38. Consider the following lines of code:

```
int i, tmp, *ptr, arr[7] = {10, 20, 30, 40, 50, 60, 70};

ptr = &arr[6];
for(i = 0; i < 4; i++) {
  tmp = arr[i];
  arr[i] = *ptr;
  *ptr = tmp;
  ptr-;
}
```

a. How many elements does the array `arr` have?
b. How would you access the middle element of `arr` and assign its value to the variable `tmp`? Do this two ways, once indexing into the array using `[]` and the other with the dereferencing operator and some pointer arithmetic. Your answer should only be in terms of the variables `arr` and `tmp`.
c. What are the contents of the array `arr` before and after the loop?

39. The following questions pertain to the code below. For your responses, you only need to write down the changes you would make using valid C code. You should verify that your modifications actually compile and run correctly. Do not submit a full C program for this question. Only write the changes you would make using legitimate C syntax.

```
#include <stdio.h>
#define MAX 10

void MyFcn(int max);
```

```
int main(void) {
  MyFcn(5);
  return(0);
}

void MyFcn(int max) {
  int i;
  double arr[MAX];

  if(max > MAX) {
printf("The range requested is too large.  Max is %d.\n", MAX);
return;
  }
  for(i = 0; i < max; i++) {
    arr[i] = 0.5 * i;
    printf("The value of i is %d and %d/2 is %f.\n", i, i, arr[i]);
  }
}
```

a. `while` loops and `for` loops are essentially the same thing. How would you write an equivalent `while` loop that replicates the behavior of the `for` loop?
b. How would you modify the `main` function so that it reads in an integer value from the keyboard and then passes the result to `MyFcn`? (This replaces the statement `MyFcn(5);`.) If you need to use extra variables, make sure to define them before you use them in your snippet of code.
c. Change `main` so that if the input value from the keyboard is between −`MAX` and `MAX`, you call `MyFcn` with the absolute value of the input. If the input is outside this range, then you simply call `MyFcn` with the value `MAX`. How would you make these changes using conditional statements?
d. In C, you will often find yourself writing nested loops (a loop inside a loop) to accomplish a task. Modify the `for` loop to use nested loops to set the ith element in the array `arr` to half the sum of the first $i-1$ integers, i.e., `arr[i]` $= \frac{1}{2}\sum_{j=0}^{i-1} j$. (You can easily find a formula for this that does not require the inner loop, but you should use nested loops for this problem.) The same loops should print the value of each `arr[i]` to 2 decimal places using the `%f` formatting directive.

40. If there are n people in a room, what is the chance that two of them have the same birthday? If $n = 1$, the chance is zero, of course. If $n > 366$, the chance is 100%. Under the assumption that births are distributed uniformly over the days of the year, write a program that calculates the chances for values of $n = 2$ to 100. What is the lowest value n^* such that the chance is greater than 50%? (The surprising result is sometimes called the "birthday paradox.") If the distribution of births on days of the year is not uniform, will n^* increase or decrease? Turn in your answer to the questions as well as your C code and the output.
41. In this problem you will write a C program that solves a "puzzler" that was presented on NPR's CarTalk radio program. In a direct quote of their radio transcript, found here

http://www.cartalk.com/content/hall-lights?question, the problem is described as follows:

> ***RAY**: This puzzler is from my "ceiling light" series. Imagine, if you will, that you have a long, long corridor that stretches out as far as the eye can see. In that corridor, attached to the ceiling are lights that are operated with a pull cord.*
>
> *There are gazillions of them, as far as the eye can see. Let us say there are 20,000 lights in a row.*
>
> *They're all off. Somebody comes along and pulls on each of the chains, turning on each one of the lights. Another person comes right behind, and pulls the chain on every second light.*
>
> ***TOM**: Thereby turning off lights 2, 4, 6, 8 and so on.*
>
> ***RAY**: Right. Now, a third person comes along and pulls the cord on every third light. That is, lights number 3, 6, 9, 12, 15, etc. Another person comes along and pulls the cord on lights number 4, 8, 12, 16 and so on. Of course, each person is turning on some lights and turning other lights off.*
>
> *If there are 20,000 lights, at some point someone is going to come skipping along and pull every 20,000th chain.*
>
> *When that happens, some lights will be on, and some will be off. Can you predict which lights will be on?*

You will write a C program that asks the user the number of lights n and then prints out which of the lights are on, and the total number of lights on, after the last (nth) person goes by. Here's an example of what the output might look like if the user enters 200:

```
How many lights are there? 200

You said 200 lights.
Here are the results:
  Light number   1 is on.
  Light number   4 is on.
  ...
  Light number 196 is on.
  There are 14 total lights on!
```

Your program `lights.c` should follow the template outlined below. Turn in your code and example output.

```
/***************************************************************************
 * lights.c
 *
 * This program solves the light puzzler.  It uses one main function
 * and two helper functions:  one that calculates which lights are on,
 * and one that prints the results.
 *
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>         // allows the use of the "exit()" function
```

```
#define MAX_LIGHTS 1000000  // maximum number of lights allowed

// here's a prototype for the light toggling function
// here's a prototype for the results printing function

int main(void) {

  // Define any variables you need, including for the lights' states

  // Get the user's input.
  // If it is not valid, say so and use "exit()" (stdlib.h, Sec 1.2.16).
  // If it is valid, echo the entry to the user.

  // Call the function that toggles the lights.
  // Call the function that prints the results.

  return(0);
}

// definition of the light toggling function
// definition of the results printing function
```

42. We have been preprocessing, compiling, assembling, and linking programs with commands like

```
gcc HelloWorld.c -o HelloWorld
```

The `gcc` command recognizes the first argument, `HelloWorld.c`, is a C file based on its `.c` extension. It knows you want to create an output file called `HelloWorld` because of the `-o` option. And since you did not specify any other options, it knows you want that output to be an executable. So it performs all four of the steps to take the C file to an executable. We could have used options to stop after each step if we wanted to see the intermediate files produced. Below is a sequence of commands you could try, starting with your `HelloWorld.c` code. Do not type the "comments" to the right of the commands!

```
> gcc HelloWorld.c -E > HW.i  // stop after preprocessing, dump into file HW.i
> gcc HW.i -S -o HW.s         // compile HW.i to assembly file HW.s and stop
> gcc HW.s -c -o HW.o         // assemble HW.s to object code HW.o and stop
> gcc HW.o -o HW              // link with stdio printf code, make executable HW
```

At the end of this process you have `HW.i`, the C code after preprocessing (`.i` is a standard extension for C code that should not be preprocessed); `HW.s`, the assembly code corresponding to `HelloWorld.c`; `HW.o`, the unreadable object code; and finally the executable code `HW`. The executable is created from linking your `HW.o` object code with object code from the `stdio` (standard input and output) library, specifically object code for `printf`.
Try this and verify that you see all the intermediate files, and that the final executable works as expected. (An easier way to generate the intermediate files is to use `gcc`

`HelloWorld.c -save-temps -o HelloWorld`, where the `-save-temps` option saves the intermediate files.)

If our program used any math functions, the final linker command would be

```
> gcc HW.o -o HW -lm          // link with stdio and math libraries, make
                                 executable HW
```

The C standard library is linked automatically, but often the math library is not, requiring the extra `-lm` option.

The `HW.i` and `HW.s` files can be inspected with a text editor, but the object code `HW.o` and executable `HW` cannot. We can try the following commands to make viewable versions:

```
> xxd HW.o v1.txt             // can't read obj code; this makes viewable v1.txt
> xxd HW v2.txt               // can't read executable; make viewable v2.txt
```

The utility `xxd` just turns the first file's string of 0's and 1's into a string of hex characters, represented as text-editor-readable ASCII characters 0..9, A..F. It also has an ASCII sidebar: when a byte (two consecutive hex characters) has a value corresponding to a printable ASCII character, that character is printed. You can even see your message "Hello world!" buried there!

Take a quick look at the `HW.i`, `HW.s`, and `v1.txt` and `v2.txt` files. No need to understand these intermediate files any further. If you do not have the `xxd` utility, you could create your own program `hexdump.c` instead:

```
#include <stdio.h>
#define BYTES_PER_LINE 16

int main(void) {
  FILE *inputp, *outputp;                          // ptrs to in and out files
  int c, count = 0;
  char asc[BYTES_PER_LINE+1], infile[100];

  printf("What binary file do you want the hex rep of? ");
  scanf("%s",infile);                              // get name of input file
  inputp = fopen(infile,"r");                      // open file as "read"
  outputp = fopen("hexdump.txt","w");              // output file is "write"

  asc[BYTES_PER_LINE] = 0;                         // last char is end-string
  while ((c=fgetc(inputp)) != EOF) {               // get byte; end of file?
    fprintf(outputp,"%x%x ",(c >> 4),(c & 0xf));   // print hex rep of byte
    if ((c>=32) && (c<=126)) asc[count] = c;       // put printable chars in asc
    else asc[count] = '.';                         // otherwise put a dot
    count++;
    if (count==BYTES_PER_LINE) {                   // if BYTES_PER_LINE reached
      fprintf(outputp,"  %s\n",asc);               // print ASCII rep, newline
      count = 0;
    }
  }
  if (count!=0) {                                  // print last (short) line
    for (c=0; c<BYTES_PER_LINE-count; c++)         // print extra spaces
      fprintf(outputp,"   ");
    asc[count]=0;                           // add end-string char to asc
    fprintf(outputp,"  %s\n",asc);          // print ASCII rep, newline
  }
  fclose(inputp);                           // close files
  fclose(outputp);
  printf("Printed hexdump.txt.\n");
  return(0);
}
```

Further Reading

Bronson, G. J. (2006). *A first book of ANSI C* (4th ed.). Boston, MA: Course Technology Press.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.