Tutorial 5 - Structs, header files and data analysis.

Welcome back to the C cave. This tutorial includes an example program built from several C source files and compiled with a Makefile. Before continuing, how did you get on with the previous challenge problem?

Challenge solution

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/sysinfo.h>

int main() {
  int i = 0;
  float ramUsed;
  char gpCmdOne[250], gpCmdTwo[250], systemCmd[1000];
  FILE *cmdPtr = 0, *outPtr = 0;
  char c, fileName[100], *strPtr = 0;
  struct sysinfo info; /* A sysinfo struct to hold the status. */

  cmdPtr = popen("hostname","r"); /* Run hostname command. */
  if(!cmdPtr) return 1;
  strPtr = &fileName[0]; /* Get a pointer to the string. */
  while((c=fgetc(cmdPtr)) != EOF) { /* Reach each character. */
    *strPtr = c; /* Set the character value. */
    strPtr++; /* Move to the next array position. */
  }
  pclose(cmdPtr); /* Close the hostname file. */
  strPtr--; /* Move backwards one array element to overwrite the new line. */
  sprintf(strPtr,"-data.txt"); /* Append the suffix. */
  printf("%s\n",fileName);
  outPtr = fopen(fileName,"w"); /* Open the output file. */
  if(!outPtr) return 1; /* If the output file cannot be opened return error */
  for(i=0;i<60;i++) {
    sysinfo(&info);  /* Get the system information */
    ramUsed = info.totalram - info.freeram;
    ramUsed /= 10240.0;
    fprintf(outPtr,"%d %f %d\n", i, ramUsed, info.loads[0]); /* Write ram used. */
    usleep(500000); /* Sleep for 1/2 a second. */
  }
  fclose(outPtr); /* Close the output file. */

  /* Now plot the data */
  sprintf(gpCmdOne, "plot \'%s\' using 1:2 title \'%s\'", fileName, "Ram used");
  sprintf(gpCmdTwo, ", \'%s\' using 1:3 title \'%s\'\n", fileName, "Load");

  /* Create the full command, including the pipe to gnuplot */
  sprintf(systemCmd,"echo \"%s%s\" | gnuplot -persist",gpCmdOne,gpCmdTwo);

  system(systemCmd); /* Execute the system command. */
  return 0; /* Return success to the system. */
}
```

The solution includes functions and techniques discussed in previous tutorials. There are more simple ways to form the file name from the host name. C provides a string.h header file which includes the declaration of several useful functions for string operations. The full list of functions can be viewed by typing man string. Strings can be concatenated by using strcat,

```
char fileName[100]="myHost", suffix[10]="-data.txt"
strcat(fileName,suffix);  /* Append suffix to fileName, result in fileName. */
```

The host name can be read using fgets rather than fgetc,

```
fgets(fileName,100,cmdPtr); /* Read until EOF or newline or 99 characters. */
```

where 100 is the size of the fileName character array. Lastly, the host name can also be read using the gethostname function from unistd.h,

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  char fileName[100], suffix[10]="-data.txt";
  gethostname(fileName,100);
  strcat(fileName, suffix); /* Append the suffix to the fileName */
  printf("%s\n",fileName);
  return 0;
}
```

Structs

Structs were introduced quickly in the last article in issue 6, to allow the use of system information. Structs occupy a continuous block of memory, similar to FORTRAN common blocks. The syntax of their usage is very similar to C++ classes with public data members. A struct is defined with a name and compound definition of variables. These variables can also be structs. Starting with a simple struct,

```
struct dataPoint {
  unsigned int timeSeconds;
  float value;
};
```

The int timeSeconds is defined first in memory and then the float value. The size of a struct in memory is the sum of the sizes of the variables. The definition of a struct should be made before its use and is typically found in a header file, but can also be written in the same file before its usage. To test this simple struct,

```
int main() {
  /* Declare a variable of "struct dataPoint" type. */
  struct dataPoint s;
  /* Assign the struct s some starting values. */
  s.timeSeconds = 60;
  s.value = 100.0;
  /* Print the size and memory locations */
  printf("sizeof(s) = %ld\n", sizeof(s));
  printf("&(s.timeSeconds) = %p\n", &(s.timeSeconds));
  printf("&(s.value) = %p\n", &(s.value));
  printf("sizeof(unsigned int) = %ld\n", sizeof(unsigned int));
  printf("sizeof(float) = %ld\n", sizeof(float));
  return 0;
}
```

where the program assigns values and prints the memory locations to demonstrate the memory structure.

When structs are passed to functions, by default a local copy of the struct is made within the function. This means that when the function finishes the value of the struct in the function which called it is unchanged. This is the same behaviour as if a basic variable had been passed to the function,

Continued over page...

```
void printDataPoint(struct dataPoint dp) {
  printf("timeSeconds = %d, value = %f\n", dp.timeSeconds, dp.value);
}
```

To modify the values of a struct within a function and retain these values, pointers can be used:

```
void clearDataPoint(struct dataPoint *dp) {
  dp->timeSeconds = 0;
  dp->value = 0.;
}
```

where the dp->timeSeconds syntax is equivalent to (*dp).timeSeconds. Other than the short hand "->" syntax, the behaviour is exactly the same as that of simple variables discussed in Issue 5.

Header files

To illustrate the use of header files, the next example defines a histogram data structure and functions. Histograms can be very useful to monitor long term experiments, provide summary figures or be used for data storage themselves.

Header files should be included to use functions within standard libraries or functions implemented within other C source files. These files contain the declaration of functions and data structures, but do not contain the implementation of the functions. The implementation is given within .c files, which are complied and built into static or dynamic libraries or directly linked to. Similar to standard header files, additional header files can be written to contain function definitions and data structures.

```
#ifndef HISTOGRAM_H
#define HISTOGRAM_H

#define MAX_BINS 1000

/* Define a data structure to hold histogram data. */
struct histogram {
  unsigned int nBins;
  float xMin;
  float xMax;
  float binContents[MAX_BINS];
};

/* Define the struct as a new type, as a shorthand. */
typedef struct histogram Histogram;

/* Fill a histogram. */
int fillHist(Histogram *, float value, float weight);

/* save a histogram to a file. */
int saveHist(Histogram *, FILE *);

#endif
```

is a header file called histogram.h, which defines a struct and declares functions, but does not implement the functions it defines. In the case that the header file is included inside another header file, the header file might be included in a program more than once. To prevent this double declaration, a precompiler ifndef case is used. This case is true the first time the header file is included and false for additional include statements. The define statements define values which are replaced when the precompiler runs, which is just before the compilation of the code. Since dynamic memory allocation has not been discussed yet, a fixed size array is used for the binContents. Lastly typedef is used to simplify the Histogram variable declaration. The header file must be included in a program before the struct or functions are used,

```
#include "histogram.h"
#include <stdio.h>
#include <stdlib.h>

int main() {
  unsigned int i;
  Histogram h; /* Create a histogram struct */
  initHist(&h,10,0.,10.); /* Initialise the histogram */
  for(i=0;i<1000;i++) { /* Generate 1000 random points */
```

```
    fillHist(&h,10*(float)rand()/RAND_MAX,1.0); /* Histogram each value. */

  }
  saveHist(&h,"Hist.txt"); /* Save the histogram. */
  return 0;
}
```

This program histograms random numbers, which are generated between zero and one. The program cannot be run without implementing functions defined in the histogram.h header file. This implementation should be given in a .c file,

```
#include "histogram.h"
#include <stdio.h>

int initHist(Histogram *hist, unsigned int nBins, float xMin, float xMax) {
  unsigned int i;
  if((hist->nBins+2) >= MAX_BINS) {
    printf("Error: too many bins requested.\n");
    return 1; /* An error has occured. */
  }
  hist->nBins = nBins;
  hist->xMin = xMin;
  hist->xMax = xMax;
  for(i=0;i<(hist->nBins+2);i++) hist->binContents[i] = 0.;
}

int fillHist(Histogram *hist, float value, float weight) {
  unsigned int ibin;
  float binSize;
  if(value < hist->xMin) ibin = 0; /* Underflow */
  else if(value >= hist->xMax) ibin = hist->nBins+1; /* Overflow */
  else { /* Find the appropriate bin. */
    ibin = 1;
    binSize = (hist->xMax - hist->xMin)/hist->nBins;
    while(value >= (ibin*binSize + hist->xMin) &&
          ibin < hist->nBins && ibin < MAX_BINS) {
      ibin++;
    }
  }
  if(ibin >= MAX_BINS) { /* Stay within the array */
    printf("Error: ibin = %u is out of range\n",ibin);
    return 1; /* An error has occured. */
  }
  hist->binContents[ibin] += weight; /* Add the weight */
  return 0;
}

int saveHist(Histogram *hist, const char *fileName) {
  FILE *outputFile = 0;
  unsigned int ibin;
  float binSize;
  outputFile = fopen(fileName, "w"); /* Open the output file. */
  if(!outputFile) { /* If the file is not open.*/
    printf ("Error: could not open %s for writing\n",fileName);
    return 1; /* An error has occured. */
  }
  binSize = (hist->xMax - hist->xMin)/hist->nBins;
  /* Write the bin centres and their contents to file. */
  ibin=0;
  while (ibin < (hist->nBins+2) && ibin < MAX_BINS) {
    fprintf(outputFile,"%lf %lf\n",
            binSize*((double)ibin+0.5) + hist->xMin,
            hist->binContents[ibin]);
    ibin++;
  }
  fclose(outputFile); /* Close output file. */
  return 0;
```

Continued over page...

Rather than type gcc several times, a Makefile can be used to build the source files and produce the executable,

```
CC = gcc
TARGET = hist
OBJECTS = $(patsubst %.c,%.o, $(wildcard *.c))

$(TARGET): $(OBJECTS)
	@echo "** Linking Executable"
	$(CC) $(OBJECTS) -o $(TARGET)

clean:
	@rm -f *.o *~

veryclean: clean
	@rm -f $(TARGET)

%.o: %.c
	@echo "** Compiling C Source"
	$(CC) -c $<
```

where more information on make is given in issue 7. Put the two .c files in the same directory as the histogram.h and Makefile. Then type make to build the executable. When the program runs it produces a text file which contains the sum of the weights within the underflow, bins and overflow. The format is chosen to allow the histogram to be plotted with gnuplot,

```
gnuplot
plot 'Hist.txt' with boxes
```

Similar to the previous tutorial, this plotting command could be added to a program also. The plot can be saved as a png file by typing,

```
set term png
set output 'Hist.png'
replot
```

after the original plotting command.

Challenge problem

Use the previous article and this article to histogram the system load for 30 minutes. Add functions to the histogram.h and histogram.c files to calculate the mean and standard deviation of the distribution. The mean of a histogram can be calculated from,

```
xMean = 0.;
for (i=1;i<=nBins;i++) { /* Skip underflow and overflow */
  xMean += binContents[i]/nBins;
}
```

The standard deviation of a histogram is given by,

```
xStdDev = 0.;
for(i=1;i<=nBins;i++) { /* Skip underflow and overflow */
  xStdDev += pow(xMean - binContents[i],2)/(nBins-1);
}
if(xStdDev > 0.) xStdDev = sqrt(xStdDev);
```

The solution to the problem will be given next time.

Article by W. H. Bell