Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.087: Practical Programming in C

IAP 2010

Problem Set 3 – Solutions

Control flow. Functions. Variable scope. Static and global variables. I/O: printf and scanf. File I/O. Character arrays. Error handling. Labels and goto.

Out: Wednesday, January 13, 2010. **Due:** Friday, January 15, 2010.

Problem 3.1

Code profiling and registers. In this problem, we will use some basic code profiling to examine the effects of explicitly declaring variables as registers. Consider the fibonacci sequence generating function `fibonacci` in `prob1.c`, which is reproduced at the end of this problem set (and can be downloaded from Stellar). The `main()` function handles the code profiling, calling `fibonacci()` many times and measuring the average processor time.

(a) First, to get a baseline (without any explicitly declared registers), compile and run `prob1.c`. *Code profiling is one of the rare cases where using a debugger like gdb is discouraged*, because the debugger's overhead can impact the execution time. Also, we want to turn off compiler optimization. Please use the following commands to compile and run the program:

```
dweller@dwellerpc:~$ gcc -O0 -Wall prob1.c -o prob1.o
dweller@dwellerpc:~$ ./prob1.o
Avg. execution time: 0.000109 msec     ← example output
dweller@dwellerpc:~$
```

How long does a single iteration take to execute (on average)?

Answer: On my 64-bit machine (results may differ slightly for 32-bit machines), the original `fibonacci()` function took 0.000109 msec on average.

(b) Now, modify the `fibonacci()` function by making the variables `a`, `b`, and `c` register variables. Recompile and run the code. How long does a single iteration take now, on average? Turn in a printout of your modified code (the `fibonacci()` function itself would suffice).

Answer: Here's the modified `fibonacci()` function for part (b):

```
void fibonacci()
{
  /* here are the variables to set as registers */
  register unsigned int a = 0;
  register unsigned int b = 1;
  register unsigned int c;
  int n;

  /* do not edit below this line */
  results_buffer[0] = a;
  results_buffer[1] = b;
  for (n = 2; n < NMAX; n++) {
    c = a + b;
```

```
    results_buffer[n] = c; /* store code in results buffer */
    a = b;
    b = c;
  }
}
```

On my 64-bit machine (results may differ slightly for 32-bit machines), the modified function took 0.000111 msec on average.

(c) Modify the `fibonacci()` function one more time by making the variable `n` also a register variable. Recompile and run the code once more. How long does a single iteration take with all four variables as register variables?

Answer: Here's the modified `fibonacci()` function for part (c):

```
void fibonacci()
{
  /* here are the variables to set as registers */
  register unsigned int a = 0;
  register unsigned int b = 1;
  register unsigned int c;
  register int n;

  /* do not edit below this line */
  results_buffer[0] = a;
  results_buffer[1] = b;
  for (n = 2; n < NMAX; n++) {
    c = a + b;
    results_buffer[n] = c; /* store code in results buffer */
    a = b;
    b = c;
  }
}
```

On my 64-bit machine (results may differ slightly for 32-bit machines), the further modified `fibonacci()` function took 3.4e-05 msec on average.

(d) Comment on your observed results. What can you conclude about using registers in your code?

Answer: The observed results suggest that storing some variables in a register vs. in memory may or may not impact performance. In particular, storing `a`, `b`, and `c` in registers do not appear to improve the performance at all, while storing `n` in a register improves performance by a factor of 3.

Problem 3.2

We are writing a simple searchable dictionary using modular programming. First, the program reads a file containing words and their definitions into an easily searchable data structure. Then, the user can type a word, and the program will search the dictionary, and assuming the word is found, outputs the definition. The program proceeds until the user chooses to quit.

We split the code into several files: `main.c`, `dict.c`, and `dict.h`. The contents of these files are described briefly below.

main.c:

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

int main() {
...
}
```

dict.c:

```
#include "dict.h"

/* data structure
for the dictionary */
char * the_dictionary[1000];

void load_dictionary() {
...
}

char * lookup(char []) {
...
}
```

dict.h:

```
/* data structure
for the dictionary */
char * the_dictionary[1000];

/* declarations */
void load_dictionary();
char * lookup(char []);
```

Answer the following questions based on the above program structure.

(a) In implementing this program, you want to access the global variable `the_dictionary` from `main.c`, as well as from `dict.c`. However, due to the header file's inclusion in both source documents, the variable gets declared in both places, creating an ambiguity. How would you resolve this ambiguity?

Answer: Adding the `extern` keyword immediately before the declaration in `dict.h` resolves the ambiguity, causing both files to reference the variable declared in `dict.c`.

(b) Now, suppose you want to restrict the dictionary data structure to be accessible only from functions in `dict.c`. You remove the declaration from `dict.h`. Is it still possible to directly access or modify the variable from `main.c`, even without the declaration in `dict.h`? If so, how would you ensure the data structure variable remains private?

Answer: Simply removing the declaration from `dict.h` does not make the variable private to `dict.c`. One could simply add an `extern` declaration to `main.c` or any other source file and still be able to access or modify the dictionary directly. In order to prevent direct access, `the_dictionary` should be declared with the `static` keyword in `dict.c`.

(c) Congratulations! You're done and ready to compile your code. Write the command line that you should use to compile this code (using `gcc`). Let's call the desired output program `dictionary.o`.

Answer: `gcc -g -O0 -Wall main.c dict.c -o dictionary.o`. Note that the order of `main.c` and `dict.c` is not important, as long as they are both specified.

Problem 3.3

Both the `for` loop and the `do-while` loop can be transformed into a simple `while` loop. For each of the following examples, write equivalent code using a `while` loop instead.

(a)
```
int factorial(int n) {
  int i, ret = 1;
  for (i = 2; i <= n; i++)
    ret *= i;
  return ret;
}
```

Answer:

```
int factorial(int n) {
  int i = 1, ret = 1;
  while (++i <= n)
    ret *= i;
  return ret;
}
```

(b)

```
#include <stdlib.h>

double rand_double() {
  /* generate random number in [0,1) */
  double ret = (double)rand();
  return ret/(RAND_MAX+1);
}

int sample_geometric_rv(double p) {
  double q;
  int n = 0;
  do {
    q = rand_double();
    n++;
  } while (q >= p);
  return n;
}
```

Note: You only need to modify the `sample_geometric_rv()` function.

Answer:

```
int sample_geometric_rv(double p) {
  double q;
  int n = 0, condition = 1;
  while (condition) {
    q = rand_double();
    n++;
    condition = q >= p;
  }
  return n;
}
```

Problem 3.4

'wc' is a unix utility that display the count of characters, words and lines present in a file. If no file is specified it reads from the standard input. If more than one file name is specified it displays the counts for each file along with the filename. In this problem, we will be implementing wc.

One of the ways to build a complex program is to develop it iteratively, solving one problem at a time and testing it throroughly. For this problem, start with the following shell and then iteratively add the missing components.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char* argv[])
{
    FILE* fp=NULL;
    int   nfiles =--argc; /*ignore the name of the program itself*/
    int   argidx =1;      /*ignore the name of the program itself*/
    char* currfile="";
    char  c;
    /*count of words,lines,characters*/
    unsigned long nw=0,nl=0,nc=0;

    if(nfiles==0)
    {
        fp=stdin; /*standard input*/
        nfiles++;
    }
    else /*set to first*/
    {
        currfile=argv[argidx++];
        fp=fopen(currfile,"r");
    }
    while(nfiles>0) /*files left >0*/
    {
       if(fp==NULL)
        {
            fprintf(stderr,"Unable to open input\n");
            exit(-1);
        }
        nc=nw=nl=0;
        while((c=getc(fp))!=EOF)
        {
          /*TODO:FILL HERE
        process the file using getc(fp)
      */
        }
        printf("%ld %s\n",nc,currfile);
        /*next file if exists*/
        nfiles--;
        if(nfiles>0)
        {
            currfile=argv[argidx++];
            fp      =fopen(currfile,"r");
        }
    }
  return 0;
}
```

Hint: In order to count words, count the transitions from non-white space to white space characters.

Answer: Here's the code with the shell filled in

```
/*
   6.087 IAP Spring 2010
   Problem set 2
*/
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char* argv[])
{
    FILE* fp=NULL;
    int   nfiles =--argc; /*ignore the name of the program itself*/
    int   argidx =1;      /*ignore the name of the program itself*/
    char* currfile="";
    char  c;
    /*following used to count words*/
    enum  state{INSIDE,OUTSIDE};
    enum  state currstate=INSIDE;

    /*count of words,lines,characters*/
    unsigned long nw=0,nl=0,nc=0;

    if(nfiles==0)
    {
        fp=stdin; /*standard input*/
        nfiles++;
    }
    else /*set to first*/
    {
        currfile=argv[argidx++];
        fp=fopen(currfile,"r");
    }
    while(nfiles>0) /*files left >0*/
    {
       if(fp==NULL)
        {
            fprintf(stderr,"Unable to open input\n");
            exit(-1);
        }
        /*TODO:FILL HERE
        process the file using getc(fp)
        */
        nc=nw=nl=0;
        while((c=getc(fp))!=EOF)
        {
            nc++;
            if(c=='\n')
            {
                nl++;
            }
            if(isspace(c))
            {
               if(currstate==INSIDE)
                   nw++;
                currstate=OUTSIDE;
            }
            else
            {
```

```
                currstate=INSIDE;
            }
        }
        /*update totals*/

        printf("%ld %ld %ld %s\n",nl,nw,nc,currfile);
        /*next file if exists*/
        nfiles--;
        if(nfiles>0)
        {
            currfile=argv[argidx++];
            fp      =fopen(currfile,"r");
        }
    }
}
```

Problem 3.5

In this problem, we will be reading in formatted data and generating a report. One of the common formats for interchange of formatted data is 'tab delimited' where each line corresponds to a single record. The individual fields of the record are separated by tabs. For this problem, download the file `stateoutflow0708.txt` from Stellar. This contains the emigration of people from individual states. The first row of the file contains the column headings. There are eight self explanatory fields. Your task is to read the file using fscanf and generate a report outlining the migration of people from Massachusetts to all the other states. Use the field "Aggr_AGI" to report the numbers. Also, at the end, display a total and verify it is consistent with the one shown below. An example report should look like the following:

```
STATE                TOTAL
-----------------------------
"FLORIDA"            590800
"NEW HAMPSHIRE"      421986
.........
-----------------------------
Total                4609483
```

Make sure that the fields are aligned.
Answer: Here's the code with the shell filled in

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

#define STRSIZE 100
#define NFIELDS 9
int main()
{
    char inputfile[]="stateoutflow0708.txt";
    /*define all the fields*/
    char state_code_org[STRSIZE];
    char country_code_org[STRSIZE];
    char state_code_dest[STRSIZE];
    char country_code_dest[STRSIZE];
    char state_abbrv[STRSIZE];
    char state_name[STRSIZE];
    char line[STRSIZE*NFIELDS];
    int  return_num=0;
    int  exmpt_num=0;
    int  aggr_agi=0;
    long  total=0;

    /*file related*/
    int fields_read=0;
    FILE* fp=fopen(inputfile,"r");
    if(fp==NULL)
    {
        fprintf(stderr,"Cannot open file\n");
        exit(-1);
    }
    /*skip first line*/
    fgets(line,STRSIZE*NFIELDS,fp);
    /*print the header*/
```

```
    printf("%-30s,%6s\n","STATE","TOTAL");
    printf("-------------------------------------\n");
    while(fgets(line,STRSIZE*NFIELDS,fp))
    {
            /*parse the fields*/
            fields_read=sscanf(line,"%s %s %s %s %s %s %d %d %d",
                               state_code_org,
                               country_code_org,
                               state_code_dest,
                               country_code_dest,
                               state_abbrv,
                               state_name,
                              &return_num,
                              &exmpt_num,
                              &aggr_agi);
            if(strcmp(state_code_org,"\"25\"")==0)
            {
             printf("%-30s,%6d\n",state_name,aggr_agi);
             total     += aggr_agi;
            }
    }
    /*print the header*/
    printf("-------------------------------------\n");
    printf("%-30s,%6lu\n","TOTAL",total);
    fclose(fp);
    return 0;
}
```

Code listing for Problem 3.1: `prob1.c`

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define NMAX 25
static unsigned int results_buffer[NMAX];

void fibonacci()
{
  /* here are the variables to set as registers */
  unsigned int a = 0;
  unsigned int b = 1;
  unsigned int c;
  int n;

  /* do not edit below this line */
  results_buffer[0] = a;
  results_buffer[1] = b;
  for (n = 2; n < NMAX; n++) {
    c = a + b;
    results_buffer[n] = c; /* store code in results buffer */
    a = b;
    b = c;
  }
}

int main(void) {

  int n, ntests = 10000000;
  clock_t tstart, tend;
  double favg;

  /* do profiling */
  tstart = clock();

  for (n = 0; n < ntests; n++)
    fibonacci();

  tend = clock();
  /* end profiling */

  /* compute average execution time */
  favg = ((double)(tend - tstart))/CLOCKS_PER_SEC/ntests;

  /* print avg execution time in milliseconds */
  printf("Avg. execution time: %g msec\n",favg*1000);
  return 0;
}
```

6.087 Practical Programming in C

January (IAP) 2010