GLG 490 Numerical Methods
Intro to C Programming – part 4

Brief introduction to pointers

You've probably already noticed that "pointers" and C go hand in hand. A pointer is simply a memory address that "points" to another memory address. This concept is one of the more difficult ones to pick up in C, however, after time and experience, the concept of pointers (and also their usefulness) begins to take firmer root. Here we just start with an introduction.

Tutorial15.c gets you acquainted with memory address. This tutorial also introduces the integer function named "sizeof()" which returns the size (in an integer number of bytes) that the argument takes up in your machine's memory.

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

/* tutorial 15 - brief introduction to pointers and sizeof() function */

int main(void)
{

/* declare an integer, a float, and a double and give them values */

int ia=4;
float fa=3.9;
double db=7.8;

/* Lets see what size (in bytes) these numbers take up in memory */
/* Note that the sizeof() function returns an integer value */

  fprintf(stdout,"an integer takes up %d bytes in this computer\n",sizeof(ia));
  fprintf(stdout,"a flaot takes up %d bytes in this computer\n",sizeof(fa));
  fprintf(stdout,"a double takes up %d bytes in this computer\n",sizeof(db));

/* Create a couple of spaces */

  fprintf(stdout,"\n\n");

/* Print out the starting address of these numbers */
/* Note: if & precedes a variable, it means we're interested in the memory address */
/* Note: %p tells the compiler to print a hexademical memory address */

  fprintf(stdout,"The starting address of variable ia is %p\n",&ia);
  fprintf(stdout,"The starting address of variable fa is %p\n",&fa);
  fprintf(stdout,"The starting address of variable db is %p\n",&db);

/* Create a couple of spaces */

  fprintf(stdout,"\n\n");

/* Now, determine the size of a pointer */
```

```
  fprintf(stdout,"pointers on this machine are %d bytes\n",sizeof(&ia));

return 0;
}
```

Tutorial16.c is similar, but introduces the idea of pointer variables. A pointer is a memory address, so we can assign a variable to a particular memory address. The memory address only gives the first byte of a number (or text), so we also need to define what data type (e.g. int, float, double, char) that the pointer refers to. This is so the compiler knows how many bytes past the starting address contains the particular value of interest. A pointer variable is declared like a normal variable except that a * is placed before the variable name (e.g. **double *my_number** declares a pointer to a double). You'll want to study *tutorial16.c* very carefully.

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

/* tutorial 16 - brief introduction to pointers variables */
/* YOU WILL WANT TO INVESTIGATE YOUR OUTPUT TO BETTER UNDERSTAND */


int main(void)
{

/* declare integers and doubles and give them values */

int kk=10;
int pp=20;

double my_number=5.3102;
double your_number=8.3;

/* now, lets declare pointer variables to doubles and  ints */

int *my_int_pointer;
double *my_double_pointer;

/* print some stuff out */

  fprintf(stdout,"Address of integer variable kk: %p  Contents: %d\n",&kk,kk);
  fprintf(stdout,"Address of integer variable pp: %p  Contents: %d\n",&pp,pp);

  fprintf(stdout,"Address of double variable my_number: %p  Contents: %f\n",&my_number,my_number);
  fprintf(stdout,"Address of double variable your_number: %p  Contents:
%f\n",&your_number,your_number);

/* The integer pointers currently do not point to anything, lets point them */

  my_int_pointer=&pp;   //we pointed the pointer to the memory address of pp

/* now lets print out stuff for "my_int_pointer */

  fprintf(stdout,"memory address that my_int_pointer points to is %p\n",my_int_pointer);
```

```
  // Note 1:  notice that we didn't have to use the & because "my_int_pointer" is already an address
  // Note 2:  upon output, you should see that "my_int_pointer" has the same address as pp

/* we can detach the pointer from pp and point it to kk instead */

  my_int_pointer=&kk;   //we pointed the pointer to the memory address of pp
  fprintf(stdout,"memory address that my_int_pointer points to is %p\n",my_int_pointer);

/* we can do the same thing with the pointer "my_double_pointer" */

  my_double_pointer=&my_number;
  fprintf(stdout,"memory address that my_double_pointer points to is %p\n",my_double_pointer);

/* Here, it gets a little confusing, but to see the contents of a pointer, we do this */

  fprintf(stdout,"the contents of the double that my_double_pointer points to is:
%f\n",*my_double_pointer);


/* lets sum it up here */
/* Note: that to make " in output, we need the \      */

  fprintf(stdout,"\n\n");
  fprintf(stdout,"We set pointer \"my_double_pointer\" to \"my_number\"\n");
  fprintf(stdout,"Pointer my_double_pointer is AT memory address %p\n",&my_double_pointer);
  fprintf(stdout,"Pointer my_double_pointer  POINTS to memory address %p\n",my_double_pointer);
  fprintf(stdout,"CONTENTS of where my_double_pointer is POINTING TO is %f\n",*my_double_pointer);

return 0;
}
```

Okay, that's enough of pointers for now. We'll return to them again later.

Functions

Functions are a key part to modular programming, and can save a lot of repetitive coding. Functions can be tricky however, particularly when passing arrays. Here we'll start with simple functions where the arguments are just simple variables (i.e. not arrays or pointers). A function is defined by the data type of the value it returns. For example a function can be a float, int, double, or char. Many functions are of type void, meaning that they don't return anything.

Some key points to remember about functions.

1) Functions must be declared properly. If they are not, serious memory trouble could occur!
2) They must be written outside of the "main()" function.
3) When variables are used as arguments, they are "copied" and are not modified inside the function. Values that are "pointed to" are a different story though! Hopefully this last part will make more sense later.
4) Functions include local variables, and these are not seen outside of the program.

Tutorial17.c is a short example of using void functions (functions that do not return a value).

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

```
#include<stdlib.h>

/* tutorial 17 - introduction to void functions */
/* carefully look at the structure of this code */


int main(void)
{

/* declarations */

double a,b;
int idummy;
char text_stuff[200];
void print_it_man();
void make_a_file();

/* assign values to the variables */
     a=10.0;
     b=5.0;
     idummy=2;
     sprintf(text_stuff,"What up?");

     print_it_man(a,b,text_stuff,idummy);

     fprintf(stdout,"Now lets check the integer: %d\n",idummy);
     fprintf(stdout,"Ah, it didn't change\n\n\n");

/* lets change text and run the function again */

     sprintf(text_stuff,"Not much?");
     print_it_man(a,b,text_stuff,idummy);

/* now lets make a file and name it file1.out */

     sprintf(text_stuff,"file1.out");

     make_a_file(text_stuff);

return 0;
}

/* FUNTION PRINT_IT_MAN */
// Things to note:
// a) a is copied to number1
// b) b is copied to number2
// etc.
// Note that the arguements are declared before the brackets!
// Note that there is NO SEMICOLON after the function name!

void print_it_man(number1,number2,text,some_integer)
    double number1,number2;
    char text[200];
    int some_integer;
{
```

```
    fprintf(stdout,"%s\n",text);
    fprintf(stdout,"number1: %f number2: %f number1 times number2 =
%f\n",number1,number2,number1*number2);
    fprintf(stdout,"the integer that was passed to this function is: %d\n",some_integer);

    some_integer=some_integer*10;

    fprintf(stdout,"we've changed that integer to be: %d\n",some_integer);


return;
}

/* FUNCTION MAKE_A_FILE */
void make_a_file(filename)
   char filename[200];
{
/* note, this function has internal variables */

FILE *fp;
int some_integer_number=20;

    if ( (fp=fopen(filename,"w"))==NULL)
    {
      fprintf(stdout,"ERROR-cant open\n");
      exit(10);
    }

   fprintf(fp,"Hi, I'm a file and here is some integer number: %d\n",some_integer_number);

   fclose(fp);

   fprintf(stdout,"We just made a file named %s\n",filename);
}
```

Now lets try an example with functions that return values. *Tutorial18.c*

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

int main(void)
{

double multiply_these_numbers();
int give_me_the_next_number();

double a=10.328;
double b=2.978;

int iman=10;
double result;
char some_text[200];

   result=multiply_these_numbers(a,b);
```

```
   fprintf(stdout,"%f times %f is %f\n",a,b,result);

   fprintf(stdout,"integer iman is %d\n",iman);
   iman=give_me_the_next_number(iman);
   fprintf(stdout,"now, it is %d\n",iman);

return 0 ;
}

/* Heres the function */
double multiply_these_numbers(number1,number2)
  double number1;
  double number2;
{

double the_result_is;

    the_result_is=number1*number2;

return the_result_is;
}

int give_me_the_next_number(kk)
  int kk;
{
int next_one;

   next_one=kk+1;

return next_one;
}
```

We'll return to functions again later.

An algorithm to help with homework#5

In homework 5, you'll need to break an interval into an integer number of pieces. You'll be given an interval along with a "preferred increment size" to break it up into to. Of course, your preferred size may not be appropriate to break that interval into an integer number of pieces, so you'll have to come up with an increment size that may be slightly smaller than the "preferred" size. In addition, you have the criteria that the number of increments must be at least 2 and that the number must be an even number. Here's how you'll do such a thing. *Tutorial19.c*

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

/* Tutorial 19 */
/* This will aid in homework #5 */


int main(void)
{
```

```
double upper_limit=5.08;
double lower_limit=1.01;
double preferred_increment_size=0.1;

double actual_increment_size;
int number_of_increments;
double total_interval_size;

   fprintf(stdout,"preferred increment size is: %f\n",preferred_increment_size);

/*
First of all, we don't always know that upper_limit is larger than lower_limit
so we should use the fabs (absolute value) function
*/

   total_interval_size=fabs(upper_limit-lower_limit);

   fprintf(stdout,"total interval size is %f\n",total_interval_size);
/*
We want to find the minimum number of intervals that will each have
a size smaller than or equal to the preferred size
To do this, we'll use the integer ceil function.  This function rounds up to
the next higher function (see your math.h sheet).
*/

   number_of_increments=ceil(total_interval_size/preferred_increment_size);

//Now, the condition that we must have at least 2 increments

  if (number_of_increments<2) number_of_increments=2;

/*
Now, to ensure that the number of increments is an even number
we use the modulus operator (remember what that does?)
If number_of_increments mod 2 has a remainder of 1, then it is
an odd number and will have to be incremented by 1
*/

  if ( (number_of_increments%2)!=0) number_of_increments++;

  fprintf(stdout,"Number of increments is %d\n",number_of_increments);

/*
now lets get the actual increment size
*/

  actual_increment_size=total_interval_size/(1.0*number_of_increments);

  fprintf(stdout,"The modified interval size is: %f\n", actual_increment_size);

return 0;
}
```

Another simple algorithm that you'll have to do in homework #5 is to multiply a prefactor to a series of numbers. The rule is that the first and last numbers have a prefactor of 1.0. For the numbers in between,

the odd numbers have a prefactor of 4.0 and the even numbers have a prefactor of 2.0. For those that are curious, we are doing Simpson rule integration for the next homework. Here's how this can be done:
Tutorial20.c

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

/* Tutorial 20 */
/* This will aid in homework #5 */


int main(void)
{

int number_of_intervals=20;
double sum;
int nint;
double prefactor;


/*
Lets some up a bunch of prefactors
where the first and last numbers have a prefactor of 1.0,
and for the other numbers,
the odd ones have a prefactor of 4.0 and the even
numbers have a prefactor of 2.0
*/

   sum=0.0;
   for (nint=0;nint<=number_of_intervals;nint++)
   {
      if (nint==0) prefactor=1.0;
      else if (nint==number_of_intervals) prefactor=1.0;
      else if ((nint%2)==1) prefactor=4.0;
      else if ((nint%2)==0) prefactor=2.0;
      else
      {
        fprintf(stdout,"There is something wrong here\n");
        exit(10);
      }

      sum=sum+prefactor;

      fprintf(stdout,"%d: prefactor: %f sum_so_far: %f\n",nint,prefactor,sum);


   }

return 0;
}
```