Homework 1: *Function Prototypes, Machine Constants, and χ^2 Critical Values*

William J. De Meo

October 7, 1997

Abstract

This paper presents programs which exhibit three fundamental programming topics. First we consider how function prototyping helps to guard against passing variables of different data types to a function. Then we evaluate machine constants, such as the largest value of each data type and the smallest floating point number in single and double precision. Finally, we construct a program which supplies the χ^2 critical value corresponding to user specified degrees of freedom and probability level.

1 Introduction

It is instructive to observe by example that specifying the data type of a function's arguments when supplying a prototype for the function will protect against the case when a variable of incorrect data type is passed to that function. In section 2.1, we construct such an example to illustrate the different results one obtains when employing different function prototypes.

In many programming contexts, knowledge of various machine constants is crucial. These constants are required when deciding whether a particular problem is ill-conditioned with respect to your machine's capabilities, as well as whether a particular algorithm to solve the problem will execute stably on your machine. For example, to address the latter concern, we might compare the error in the computed solution to the *machine epsilon*. We define machine epsilon as the smallest number such that, when added to 1, the result is greater than 1. Intuitively, we think of this as the smallest non-zero number. Since this corresponds to the accuracy with which the machine can represent a small error, we can't expect the error in our algorithm to be much smaller than machine epsilon. On the other hand, an error much larger than machine epsilon, calls the algorithm's stability into question.

Other numbers which are often required when deciding whether a problem can be stably solved on your machine are the largest integer (both long and short) which can be handled by the machine, the largest unsigned integer, and the largest floating point number (both single and double precision). In section 2.2, we will construct a program which estimates each of these, give results for the Sun Ultra 2, and demonstrate partial failure of the program on an Intel Pentium processor.[1]

In statistical applications, it is often necessary to produce a critical value for a given probability distribution at a given level of significance. In section 2.3, we present a program which provides the critical value for the χ^2 distribution with user specified degrees of freedom and level of significance.

[1]using the gcc compiler under the Linux operating system.

2 Programs

2.1 Function Prototypes

The program listing demonstrating the use of two different prototypes appears in the appendix. Some output from running this program appears below.

First trial:

```
% Sum
No valid argument passed to Sum.

Enter first integer:  2

Enter second integer:  5

The sum of 2 and 5 is -2147483648.
```

Second trial:

```
% Sum
No valid argument passed to Sum.

Enter first integer:  3

Enter second integer:  2

The sum of 3 and 2 is -2147483648.
```

Third trial:

```
% Sum -s
Argument 1:  -s
Entering Smart Mode...

Enter first integer:  2

Enter second integer:  5

The sum of 2 and 5 is 7.
```

Fourth trial:

```
% Sum s
Argument 1:  s
Entering Smart Mode...

Enter first integer:  3
```

```
   Enter second integer:  2

   The sum of 3 and 2 is 5.
```

As the output makes clear, when the program is invoked with no option, the answer is wrong. The function prototype being used in this case is

```
   double sum();
```

When the `-s` options is used, the prototype is

```
   double sum(double, double)
```

which has the effect of automatically casting the integer arguments to double before evaluating the function. Thus we get the correct results.

2.2 Machine Constants

The listing for the program mconstants.c appears in the appendix, along with it's output immediately following. To summarize that output, we see that the single precision machine epsilon is about 6.27583e-08. The double precision machine epsilon is about 1.15829e-16. The largest short and long are 16384 and 1073741824, respectively. The largest unsigned short and unsigned long are 32768, 2147483648, respectively. Finally, the largest single and double precision float are 1.70141e+38 and 8.98847e+307, respectively.

2.3 χ^2 Critical Values

A program which computes the critical values of the chi-square distribution, with a user specified degrees of freedom and probability level, is listed in the appendix. Some sample output appears below.

```
% chisquare

   Enter the left tail probability level:  .2

   Enter the degrees of freedom:  15

   The corresponding chi-square value is:  10.306963

   That is, P( Chi-square_(15) < 10.306963 ) = 0.200000

   Press X to exit, or any other character to continue:  y

   Enter the left tail probability level:  .3

   Enter the degrees of freedom:  15

   The corresponding chi-square value is:  11.721176

   That is, P( Chi-square_(15) < 11.721176 ) = 0.300000
```

```
Press X to exit, or any other character to continue:  y

Enter the left tail probability level:  .5

Enter the degrees of freedom:  10

The corresponding chi-square value is:  9.341823

That is, P( Chi-square_(10) < 9.341823 ) = 0.500000

Press X to exit, or any other character to continue:  X
```

3 Discussion and Conclusions

The programs appearing in the appendix seem to give reasonable and useful output, and answer the questions we posed to begin this paper. In particular, we have seen how a function fails when, expecting arguments of data type `double`, it instead receives arguments of data type `int`. We have also seen how a more precise function prototype can guard against this type of failure. The `mconstants.c` showed us the important machine constants for the Sun Ultra 2. Finally, the `chisquare.c` program computed critical values from the χ^2 distribution.

4 Appendix

Contents

- Program listing for: `sum.c`
- Program listing for: `mconstants.c`
- Program output of: `mconstants.c`
- Program listing for: `chisquare.c`

4.1 Program listing for: `sum.c`

```
/*----------------------------------------------------------------------
  sum.c

  Created by William J. De Meo
    on 9/27/97

  This program adds two double precision floating point numbers.
  and was written to see what happens when such a function
  receives integer arguments.

  The "dumb" version of the function is the default.  It expects double,
  and only double, precision floating point numbers.  If the -s option
  is specified, the "smart" version of the function is called, which gives
  the correct sum for integers, single floats, double floats, etc.
---------------------------------------------------------------------*/
#include <string.h>

main(int argc, char *argv[]) /* Possible arguments:        */
{ /* -s or s to invoke smart version */
     double sum();
     double ssum(double, double);

     int x,y,sflag=0;

     if(argv[1])
     {                              /* strcmp() returns zero if strings are equal */
  if(!strcmp(argv[1],"-s") || !strcmp(argv[1],"s"))
  {
       sflag=1;
       printf("Argument 1: %s\nEntering Smart Mode...\n",argv[1]);
  }
     }
     else printf("No valid argument passed to %s.\n",argv[0]);

     printf("\nEnter first integer: "); scanf("%d",&x);
     printf("\nEnter second integer: ");scanf("%d",&y);

     if(sflag) printf("\nThe sum of %d and %d is %d.\n",x,y,(int)ssum(x,y));
     else       printf("\nThe sum of %d and %d is %d.\n",x,y,(int)sum(x,y));
}

double sum(double x, double y)
{
     return(x+y);
}

double ssum(double x, double y) {sum(x,y);}
/* ssum() gets the same function definition */
```

4.2 Program listing for: `mconstants.c`

```
/*---------------------------------------------------------------------
  mconstants.c

  Created by William J. De Meo
    on 9/27/97

  This program's purpose is to find various machine parameters.
  (All numbers have been type casted in an attempt to prevent the compiler
  from forcing higher precision.)
----------------------------------------------------------------------*/
#include <stdlib.h>

main()
{

  short S, bigS;
  long L, bigL;
  unsigned short US, bigUS;
  unsigned long UL, bigUL;
  float F, bigF;
  double D, bigD;
  float delta, eps, negeps;
  double ddelta, deps, dnegeps;


  /* First let's check out the size of each data type on this machine */
  printf("\n      short: %d bytes.",sizeof(short));
  printf("\n        int: %d bytes.",sizeof(int));
  printf("\n       long: %d bytes.",sizeof(long));
  printf("\n      float: %d bytes.",sizeof(float));
  printf("\n     double: %d bytes.",sizeof(double));
  printf("\nlong double: %d bytes.",sizeof(long double));
printf("\n---------------------------------------------------------------------\n");

  /* Find Largest Short */
  S = (short) 1;
  do
    {
      bigS = S;
      S *= (short) 2;
    }while(bigS == (S / (short) 2));   /* If S*2 too big for machine, then  (S*2)/2 != S */
                            /* and loop exits.  S stores the spurious (oversized) value.  */

  /* Find Largest Long */
  L = (long) 1;
  do
    {
      bigL = L;
      L *= (long) 2;
    }while(bigL == (L / (long) 2));


  /* Find Largest Unsigned Short */
  US = (unsigned short) 1;
  do
    {
      bigUS = US;
      US *= (unsigned short) 2;
    }while(bigUS == (US / (unsigned short) 2));

  /* Find Largest Unsigned Long */
  UL = (unsigned long) 1;
  do
    {
      bigUL = UL;
      UL *= (unsigned long) 2;
    }while(bigUL == (UL / (unsigned long) 2));

  /* Find Largest Single Precision Float */
  F = (float) 1;
  do
    {
      bigF = F;
      F *= (float) 2;
    }while(bigF == (F / (float) 2));

  /* Find Largest Double Precision Float */
  D = (double) 1;
  do
    {
      bigD = D;
      D *= (double) 2;
    }while(bigD == (D / (double) 2));


  /*** MACHINE EPSILON ***/

  /* Find Smallest Single Precision Float */
  /* Two methods produce different results */
  /* The results are related by:  eps = 2 * negeps  */


  /* First method: test that addition to 1 is greater than 1 */
for(delta= (float) 1 ;(float) 1 + delta > (float) 1;)
    {
      eps=delta;
      delta /= (float) 1.1;
    }

  /* Second method: test that subtraction from 1 is less than 1 */

  for(delta= (float) 1;(float)((float) 1 - delta) < (float) 1;)
    {
      negeps=delta;
      delta /= (float) 1.1;
    }

  /* Find Smallest Double Precision Float (agian by two methods) */
  ddelta = (double) 1;
  while((double) 1 - ddelta < (double) 1)
    {
      dnegeps=ddelta;
      ddelta /= (double) 1.1;
    }

  for(ddelta = (double) 1; (double) 1 + ddelta > (double) 1; )
    {
      deps=ddelta;
      ddelta /= (double) 1.1;
    }

  printf("\n                      largest short and long:  %d, %d",bigS, bigL);
  printf("\n    largest unsinged short and unsigned long:  %u, %u",bigUS, bigUL);
  printf("\n   largest single and double precision float:  %g, %g",bigF, bigD);
  printf("\nsingle precision machine epsilon and neg-eps:  %g, %g",eps, negeps);
  printf("\ndouble precision machine epsilon and neg-eps:  %g, %g",deps,dnegeps);
  printf("\n-----------------------------------------------------------------------\n");
  printf("\nValues of Variables After Overflow");
  printf("\n-----------------------------------------------------------------------\n");
  printf("                     short and long:  %d, %d",S,L);
  printf("\n unsinged short and unsigned long:  %u, %u",US, UL);
  printf("\nsingle and double precision float:  %g, %g",F, D);
  printf("\n-----------------------------------------------------------------------\n");
  printf("\nValues of Variables After Underflow");
  printf("\n-----------------------------------------------------------------------\n");
  printf("  single precision machine eps:  %g",delta);
  printf("\ndouble precision machine eps:  %g",ddelta);
  printf("\n-----------------------------------------------------------------------\n");

printf("\n\nProof that computed epsilons are not zero:");
  printf("\n-----------------------------------------------------------------------\n");
printf("\nSingle precision (25 decimal display):");
printf("\n   1 + eps = %.25e",(double) 1 + eps);
printf("\n1 - negeps = %.25e\n", (double) 1 - negeps);
printf("\n\nDouble precision (25 decimal display):");
printf("\n   1 + eps = %.25e",(double) 1 + deps);
printf("\n1 - negeps = %.25e\n", (double) 1 - dnegeps);
  printf("\n-----------------------------------------------------------------------\n");
}
```

4.3 Program output of: `mconstants.c`

```
  /*******************************************/
 /**  Output from the mconstants.c program  **/
/*******************************************/


      short: 2 bytes.
        int: 4 bytes.
       long: 4 bytes.
      float: 4 bytes.
     double: 8 bytes.
long double: 16 bytes.
----------------------------------------------------------------------

                      largest short and long:  16384, 1073741824
    largest unsinged short and unsigned long:  32768, 2147483648
   largest single and double precision float:  1.70141e+38, 8.98847e+307
single precision machine epsilon and neg-eps:  6.27583e-08, 3.22049e-08
double precision machine epsilon and neg-eps:  1.15829e-16, 5.94384e-17
----------------------------------------------------------------------

Values of Variables After Overflow
----------------------------------------------------------------------
                     short and long:  -32768, -2147483648
 unsinged short and unsigned long:  0, 0
single and double precision float:  Inf, Inf
----------------------------------------------------------------------

Values of Variables After Underflow
----------------------------------------------------------------------
  single precision machine eps:  2.92772e-08
double precision machine eps:  1.05299e-16
----------------------------------------------------------------------


Proof that computed epsilons are not zero:
----------------------------------------------------------------------

Single precision (25 decimal display):
   1 + eps = 1.0000000627583176537882537e+00
1 - negeps = 9.9999996779506261646019993e-01


Double precision (25 decimal display):
   1 + eps = 1.0000000000000002220446049e+00
1 - negeps = 9.9999999999999988897769754e-01

----------------------------------------------------------------------
```

4.4 Program listing for: `chisquare.c`

```
/*----------------------------------------------------------------------
  chisquare.c

  Created by William J. De Meo
    on 10/7/97

    Purpose:  This program obtains degrees of freedom (df) and
    left tail probability level (p) from the user,
    generates the corresponding Chi-square value, and awaits another query.

    Notes: On the SCF network, type xnaghelp for documentation regarding
           the NAG subroutine g01fcf_
----------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>

void main()
{
     double df,p,chi;
     char another='y';
     int ifail; /* -1 instructs routine to */
     /* report errors and continue */

     double g01fcf_(double *p, double *df, int *ifail);

     while(((another!='x') && (another!='X')) || another=='\0')
     {
  printf("\nEnter the left tail probability level: "); scanf("%lf",&p);
  printf("\nEnter the degrees of freedom: "); scanf("%lf",&df);

  ifail = -1;
  chi = g01fcf_(&p,&df,&ifail);

  if(ifail != 0)
  {
      switch(ifail){
      case 1:
    printf("\nProbability level out of range [0,1).");
    break;
      case 2:
    printf("\nDegrees of freedom not greater than 0.");
    break;
      case 3:
    printf("\nProbability too close to 0 or 1 for value to be calculated.");
    break;
       case 4:
    printf("\nSolution failed to converge. Result is only an approximation.");
    printf("\nThe corresponding chi-square value is: %lf\n", chi);
    printf("\nThat is, P( Chi-square_(%lf) < %lf ) = %lf", df,chi,p);

    break;
       case 5:
    printf("\nSolution failed to converge.");
    break;
       default:
    break;
       }
  }
  else
  {
       printf("\nThe corresponding chi-square value is: %lf\n", chi);
       printf("\nThat is, P( Chi-square_(%d) < %lf ) = %lf", (int)df,chi,p);
  }
  printf("\n\nPress X to exit, or any other character to continue: ");
  scanf("%1s",&another);
     }
}
```