A Pedagogically Significant Approach to Inputting Mathematical Expressions at Runtime in C

V. N. Krishnachandran, Jisha Jose Panackal and Salkala K.S.
Vidya Academy of Science & Technology,
Thrissur - 680501, Kerala, India.

ABSTRACT
The problem of inputting a mathematical expression at runtime in C is generally considered very difficult. The general approach to solve the problem is to include specialized parser packages as header files. It appears that such packages are available only for advanced versions of C like C++ or C#. In this paper we consider a very elementary solution to the problem which does not make use of external packages and uses only the basic concepts of the C programming environment. It also does not require the in-line implementation of the extremely difficult task of parsing a mathematical expression. This approach is accessible to beginning programmers also. In broad outline, the approach adopted is that while a programme is being executed, it generates another programme, compiles and executes the new programme, and finally returns to the original programme. Even though the method may not be satisfactory in terms of speed or efficiency, it is pedagogically significant as it can be employed as a tool for throwing more light on the basic concepts of compilation and execution of a programme.

Keywords – mathematical expressions; runtime evaluation; C; eval; inputting mathematics; arser

1. INTRODUCTION
Every school boy or girl who had some experience with computer programming would have written programmes to compute and print the values of mathematical expressions. For example, every school child would be capable tackling the elementary problem of writing a programme to compute and print the amount of simple interest using the formula for calculating the simple interest, or the advanced problem of computing and printing the length of the hypotenuse of right angled triangle given the lengths of the other two sides of the triangle (advanced because it uses the mathematical function for the extraction of square root of a given number). In these methods, the child is forced to write a new programme to evaluate every new mathematical expression. In this paper we examine whether we can specify the function to be evaluated at runtime so that there would be one universal programme which can be used to compute and print the values of any given mathematical expression.

We have explored this possibility in this paper in the context of the C language. Every C programmer knows how to compute and print the value of a mathematical expression. The expression to be evaluated has to be encoded in the syntax of the C language and has to be placed at the appropriate location in the program. The expression can contain any of the built-in functions known to C, or, it may also contain suitable user defined functions. The C code for a mathematical expression is a string of characters and so every mathematical expression can obviously be represented as a string. The problem that we are posing to ourselves in this paper, is how to specify this string at runtime. More clearly, the problem we are examining is whether it is possible to create a programme which accepts a mathematical expression at runtime as a string, then compute and print the value of the expression. In other words, we need a programme which evaluates, or coverts, a string into a mathematical expression.

A moment's thought would be enough to convince oneself that the requirement put forward in the previous paragraph is not an idly speculated need at an elementary level. A situation like this arises if one tries to write one universal program which can be used for numerically solving any given equation f (x) = 0 using standard techniques like the bisection method or the Newton-Raphson algorithm. We need one program which, while being executed, accepts any function f (x) as in-put and prints the solution of the equation f (x) = 0. The necessity, or rather the possibility, of writing such a programme is interesting in itself and will be an illuminating learning experience for students learning the C programming language. This problem is generally considered as a difficult problem in the context of the C language. But it is trivial in some programming environments. In Section 2 of this paper, we have presented a few programming languages which provide an easy solution to the problem presented above. This is not an exhaustive analysis of all programming contexts. Rather, it is a sample of the available techniques. In later sections we examine how the problem can be solved in C environment.

In C or C++ there is no direct facility for the run-time evaluation of expressions ([1], Section 5.7.3.). In the C context, most programmers suggest a method wherein the mathematical expressions are parsed either within the programme, or use specialized header files which simplifies the task of parsing. Preparing the code for parsing from scratch, though possible, is error prone and not very satisfactory. In this paper we show that there is indeed a very elementary solution to the problem which is very easy to understand and implement. The basic idea of the suggested solution to the problem is to write a programme which at the execution time creates a second programme, The second programme is compiled and executed while the first programme is still running. After completing the execution of the second programme, the control goes back to the original programme and the original programme is terminated. The steps leading to the solution are explained in Section 3. The outline of the proposed solution is discussed in more detail in Section 4. The solution is presented in Section 5. The paper concludes with a discussion of the pedagogical significance of the ideas presented in the paper.

The programmes and their outputs relating to the proposed solution are presented in the environment of the GNU compiler collection. This has certain advantages in relation to invoking the compiler. the ideas presented could however be used to develop a solution of the problem in the context of other C compilers like Borland Turbo C, etc.

2. INPUTTING MATHEMATICAL EXPRESSIONS AT RUNTIME IN OTHER LANGUAGES
In this section we discuss how the problem of inputting mathematical expressions at runtime is solved in a few popular languages. It may be noted that the first language to introduce this feature was Lisp [12] and it was implemented by defining the eval statement. This feature was subsequently adopted in nearly all later scripting languages. However systems programming languages like C lack a similar facility [1]. We have briefly discussed how eval is implemented in BASIC, Perl, Python and JavaScript. The fact that Visual Basic, a language that allows programmers to create simple

as well as complex GUI applications, has the capability for accepting mathematical expressions at runtime is also noted below. This is achieved via the use a specially designed ActiveX control (see [1] pp.81 83).

A. BASIC

The need for a facility for inputting mathematical expressions at runtime had been felt even at the early stages in the development of high level programming languages. The BASIC was a high-level programming language developed in 1964 with the declared purpose of providing computer access to non-science students. Some later versions of BASIC had included the eval statement. With the help of this statement, a programme could accept any mathematical expression such as sin(x) + cos(x) at runtime and print the value of the expression for any specified value of x ([1] p.66). 'BBC BASIC' is one version of BASIC which had implemented such a feature. The following programme in BBC BASIC evaluates the expression typed in by the user [2].

```
PRINT This program evaluates the
expression
PRINT you type in and prints the answer
REPEAT
   INPUT Enter an expression
   exp$
   IF exp$<>END PRINT EVAL
   exp$

UNTIL
exp$=END
END
```

B. Perl

Perl, a language developed in the late 1980s, though basically designed for text processing, has a feature for run-time evaluation of expressions and this feature is implemented through the use of the eval statement. In Perl this is a tremendously powerful and popular statement and it is sometimes touted as one significant feature that sets it apart from the C language.[3] Here is a sample code in Perl using eval ([1], p.67):

```
# Put some code inside \$str
\$ str = $c=$a+$b;

\$ a   = 10 ; $b = 20;  eval \$ str ;

print  \$  c  ;
```

C. Python

Python language can also evaluate expressions on the fly. Again, the feature is implemented via the eval statement [6]. In Python, we can pass a chunk of text to a running Python application, and it is parsed, compiled and executed on the fly. In Python, the interpreter is always available. For example consider the problem: Given a mathematical expression in the language as a string with a free variable named x let it be required to evaluate it with x bound to a provided value, then evaluate it again with x bound to another provided value, then subtract the result of the first from the second and print it. The Python solution to the problem is given below:

```
>>> def evalwithx(code, a, b):
      return
          eval(code,{'x':b})-
              eval(code,{'x':a})

>>> evalwithx('2 ** x', 3, 5)
24
```

D. JavaScript

The JavaScript solution to the problem discussed under Python is presented below.

```
Function evalWithX(expr, a, b)
{
  var x  =  a;
  var atA  = eval(expr);
  x = b;
  var atB  =  eval(expr);
  return  atB atA;
```

E. Visual Basic

In Visual Basic the facility for runtime evaluation of expressions can be implemented by making use of the ActiveX control known as the Script control. This control comes with VB6 and all later versions of Visual Basic.It can be downloaded from http://msdn.microsoft.com /scripting (see [4], [5]).

3. INITIAL STEPS TO THE PROPOSED NEW SOLUTION

The ideas leading to our solution to the problem of inputting mathematical expressions at runtime is best understood by analyzing a simple problem. These initial steps are discussed in this section. To this end, we consider the following elementary problem:

"Compute and print the values of the expression x*x+x+1 for values of x form 1 to 10 in increments of 2 in the form a table."

Program 01 accomplishes the task. The sample output is also given.

Program 01

```
#include<stdio.h>
main()
{
float x; printf("\tx\t\tx*x+x+1\n"); for(x=1; x<=10; x=x+2)
{
printf("\t%f\t\t%f\n",x,x*x+x+1);
}
}
```

Program 01 : Output

x	x*x+x+1
1.000000	3.000000
3.000000	13.000000
5.000000	31.000000
7.000000	57.000000
9.000000	91.000000

We now reformulate the code in Program 01 as in Program 02 below. Note the use of the preprocessor directive define.

Program 02

#include<stdio.h> #define F x*x+x+1; #define A 1.0 #define B 10.0 #define H 2.0 main()

{

float x; printf("\tx\t\tF\n"); for(x=A; x<=B; x=x+H)
{
printf("\t%f\t\t%f\n", x, F);
}
}

The four define statements in Program 02 are next put in a separate file named definitions.c (see Program 03).

Program 03 (`definitions.c`)

#define F x*x+x+1; #define A 1
#define B 10
#define H 2

The code in Program 02 is further modified by replacing the four define statements with a single include directive to include the file

`definitions.c`. This modified code is given in Program 04. We call the file containing this program background.c.

Program 04 (`background.c`)

```
#include<stdio.h>
#include"definitions.c"
main()
{
    float x;
    printf("\tx\t\t
    F\n"); for(x=A;
    x<=B; x=x+H)
    {
        printf("\t%f\t\t%f\n",  x,  F);
    }
}
```

We next consider a program which will generate the file `definitions.c`. While generating this file we input the expression `x*x+x+1` as a string. The necessary code is given as Program 05. A sample output while running the program is also given.

Program 05

```
#include<stdio.h>
main()
{
   FILE *fun;
   char xx[100];

   char aaa[20], bbb[20], hhh[20];
   int i;
   for(i=0; i<100; i++){
     xxx[i]=\0; }
   for( i=0; i<20; i++ )
     {aaa[i]=\0; bbb[i]=\0;
      hhh[i]=\0;}

   fun=fopen("definitions.c","w");
   system("clear");
   printf("\n  Enter f(x)");
   scanf("%s",&xxx);
   fprintf(fun,"#define F  ");
   fprintf(fun,"%s",xxx);
   fprintf(fun,"\n");
   printf("\n Enter minimum of x:");
   scanf("%s",&aaa);
   fprintf(fun,"#define A  ");
   fprintf(fun,"%s",aaa);
   fprintf(fun,"\n");
   printf("\n Enter maximum of x:");
   scanf("%s",&bbb);
   fprintf(fun,"#define B  ");
   fprintf(fun,"%s",bbb);
   fprintf(fun,"\n");
   printf("\n Enter incr. in x: ");
   scanf("%s",&hhh);
   fprintf(fun,"#define H  ");
   fprintf(fun,"%s",hhh);
   fprintf(fun,"\n");
   fclose(fun);
}
```

```
Program 05 : Sample output 1
Enter f(x):x*x+x+1
Enter minimum of x:1
Enter maximum of x:10
Enter incr. in x:2
```

After completion of the execution of Program 05, the programme would have created a file named definitions.c with contents as in Program 03.

4. DESCRIPTION OF THE METHOD

An analysis of the process described in the previous section will show that the procedure involves two steps:

1. Compilation and execution of the program in Program 05. This generates the file `definitions.c`.
2. Compilation and execution of `background.c` given in Program 04. This generates the required output.

The crucial idea in the present method of inputting mathematical expressions at run time is the possibility of creating a program which incorporates the above two steps. To do this we compile and execute background.c, the activity in the second step, while executing the program in Program 05. This can be accomplished by invoking the system command in two stages.

We have to invoke the compilation programme to accomplish the task. The way the compilation programme is invoked depends on the particular compiler being used. The GNU C compiler (GCC) widely available with Linux distributions is discussed here. This particular compiler is simple to apply in the present circumstances. The standard directory structure associated with GNU/Linux distributions makes accessibility of the compiler a simple matter.

5. IMPLEMENTATION

In the first stage, the program `background.c` is compiled

```
printf("\n  Enter  f(x):");
scanf("%s",&xxx);
fprintf(fun,"#define F  ");
fprintf(fun,"%s",xxx);
fprintf(fun,"\n");
printf("\n  Enter minimum of  x:  ");
scanf("%s",&aaa);
fprintf(fun,"#define A  ");
fprintf(fun,"%s",aaa);
fprintf(fun," \n");
```

using the gcc compiler. Assuming that one is working in a Linux operating system environment we may use the GNU C compiler to accomplish this task by including the following line of code:

```
system("cc lm w o x.out ./background.c");
```

The switch `lm` is used to make available the functionalities and libraries of the math.h header file. The switch `w` will suppress any warning messages that may be generated at the time of compilation. The output file will be named `x.out`.

In the second stage, the output file `x.out` is executed using the following code (again assuming a Linux environment):

```
system("./x.out");
```

Before exiting the execution, a little cleaning up can be performed. The files `definitions.c` and x.out may be deleted. The program `mathinput.c` given in Program 06 is derived from the program given in Program 05 by incorporating the two changes suggested above.

Program 06 (`mathinput.c`**)**

```
#include<stdio.h>
main()
{
//  Defining the variables FILE
*fun;

FILE *back; char
xx[100];

char aa[20],bbb[20],hhh[20]; int I;

//  Initialising the strings
for(i=0;i<100;i++){xxx[i]=\0;}
for( i=0; i<20; i++ )
{aaa[i]=\0;bbb[i]=\0;hhh[i]=\0;}

//  Creating the file background.c
back=fopen("background.c","w");
fprintf(back,"#include<stdio.h>\n");
fprintf(back,"#include \
\"definitions.c\"\n");
fprintf(back,"main()\n"); fprintf(back,"{
\n"); fprintf(back," float x;\n\n");
fprintf(back, \" printf (\"\\t\\tx \\t
\\t\\t F\\n\");\ \n");
fprintf(back,  \" for(x=A; x<=B; x=x+H)
\n"); fprintf (back,"{\n"); fprintf(back, \"
printf("\t\t %cf\t\t%cf \n", x, F); \
\n",37,37);

fprintf(back,"}\n");
fprintf(back,"}\n");
fclose(back);

//  Opening  the  file  definitions,c
fun=fopen("definitions.c","w");

//  Cleaning  up  the  monitor
system("clear");
//  Inputting  the  mathematical
//  expression  and  constants
//  and  writing  these  in
//  the  file  definitions.c
printf("\n  Enter maximum  of x:  ");

scanf("%s",&bbb);
fprintf(fun,"#define B ");
fprintf(fun,"%s",bbb);
fprintf(fun," \n");
printf("\n Enter increment in x:");
scanf("%s",&hhh); fprintf(fun,"#define H
"); fprintf(fun,"%s",hhh);
fprintf(fun," \n");
fclose(fun);

system("cc w lm -o  x.out \
./background.c");
system("./x.out");

//      Cleaning  up;

//      removing programme created files.
   system("rm ./x.out");

system("rm ./definitions.c");
system("rm ./background.c");
}
```

Program 06 : `Sample output`

```
Enter   f(x)             :  (exp(x)+exp(-x))/2.0
Enter  minimum  of  x    :  0
Enter  maximum  of  x    :  1
Enter  increment in x :     .2
x                           f(x)
0.0000                      1.0000
0.2000                      1.0201
0.4000                      1.0811
0.6000                      1.1855
0.8000                      1.3374
1.0000                      1.5431
```

6. CONCLUSION

This paper is concerned with writing a programme in C for evaluating a mathematical expression supplied at the time of the execution of the programme. We have explored how this problem is solved in a few other languages. We have observed that the generally accepted solution to the problem is to use specialised expression parsers. In this paper, however, we have presented an elementary solution to the problem and illustrated it by presenting a sample solution using the GNU compiler collection. The solution is not fully satisfactory because it involves the invocation of the compiler during the execution of the programme. Even though the solution is not satisfactory, it illustrates the possibility of a solution and this is pedagogically very illuminating. It helps to highlight the use of the system command in the C command repertoire. More work needs to be done to implement the facility for runtime

evaluations of mathematical expressions in the C programming environment.

Acknowledgement. A preliminary version of paper was presented in the 2nd International Conference on Information Systems and Technology, held at MES College of Engineering, Kuttippuram, Kerala, India during 16, 17 May 2011. The authors acknowledge with thanks the organizers of the Conference for giving them an opportunity to present the paper in the Conference. The authors are also grateful to the participants of the Conference whose many constructive comments have helped to develop the paper to the present form.

7. REFERENCES

[1] Sriram Srinivasan, *Advanced Perl programming*, OReilly Media, Inc., 1997.

[2] Website dedicated to BBC BASIC: http://www.bbcbasic.co.uk/bbcbasic.html

[3] Steven Holzner, *Perl Black Book (2nd Ed.*) Dreamtech Press, 2004 (p.224).

[4] George Shepherd, "Add Scripting to Your Apps with Microsoft ScriptControl", MSDN Magazine, June 2000. Available: http://msdn.microsoft.com/ en-us/ magazine/ cc302278.aspx

[5] Evangelos Petroutsos, Richard Mansfield, *Visual Basic .NET Power Tools*, John Wiley and Sons, 2003 (p.463 464).

[6] Python v.2.7.1 documentation: The Python Standard Library:2. Built-in Functions". Available: http://docs.python.org/library/functions.html#eval

[7] TbcParser Math Expression Parser 1.01 . http://www.vclcomponents.com/ Delphi/Components Collection/ TbcParser Math Expression Parser-info.html

[8] Marcin Cuprjak, "'Evaluating Mathematical Expressions byCompiling C# Code at Runtime", April 2003.Available:http://www.codeproject.com /KB/recipes/matheval.aspx

[9] A Function to Evaluate Arithmetic Expressions. Available:http://www.parsifalsoft.com/examples/ evalexpression/ index.htm

[10] John McCarthy, Stanford University,"History of Lisp." February 1979. Available: http://www-formal.stanford.edu/jmc/history/lisp/node3.html