Crossref
Similarity Check
Powered by iThenticate

Reason- A Technical Journal
ISSN 2277-1654
Volume XXII 2023

REVISITING THE PROBLEM OF VARIABLE NUMBER OF LOOPS IN PROGRAMMING IMPLEMENTING A 'CODE GENERATOR' METHOD

P. Mukhopadhyay[1] and P. Mukhopadhyay[2]

[1]Department of Mathematics, Asansol Engineering College, Asansol-713305, India.
Email: parthamukhopadhyay1967@gmail.com
[2]Department of Computer Science, University of Calcutta, Salt Lake, West Bengal, India.
Email: puspaulmukhopadhyay@gmail.com

Paper submitted on June 11 2023, Accepted after revision on July 07 2024
DOI : 10.21843/reas/2023/1-7/47344

Abstract: Some algorithms require a variable number of loops to be used within a program and the number depends on user input during the runtime. For example, generating all possible arrangements of n given different symbols is one such problem where the thing **variable number of loops is needed as the number n may vary at the runtime**. Changing the raw code and recompiling it for every different user input is simply absurd. In the present **proposed** work, the authors have tried to find a work around by using the code generating technique by running a parent code. Code generated by a parent code-generator is generally not preferred in situations where it is supposed to be subsequently edited by human intervention in regular frequency. Some systems such the wizards for installing application software like VC++ etc. generate code which the programmer is expected to edit by hand. This cannot be popular by any common sense. The authors have devised a method to avoid any human intervention in between. Generated code should be saved in a separate file and then be compiled to generate the final desired piece of code for the desired output.

Keywords: Distinct arrangements of symbols, variable depth loops, code generating code.

1. INTRODUCTION

Some algorithms (vide [1, 2]) require a variable number of loops to be used within a program and the number depends on user input during the runtime. There are many problems where the number of loops may vary on user input or input from any other software system as a part of process. Generating all possible arrangements of n different symbols, selecting all possible subsets from a given set (i.e., power set of a finite set) with a cardinality n which may vary are such common problems where the above thing is needed. Implementing variable number of loops in a program based on an input in the runtime has been an intriguing problem in programming. Any student of algorithm and programming is expected to be familiar with the treatise of Gottfried [3] on introductory C (Schaum's series). In this book, one example of coding is given to generate all distinct arrangements of three consecutive integers say {1, 2, 3}. The logic is simple but ingénues! Here is the essential piece of code in C:

```
.....
.....
for( x =1;x <=3;x++) {
        for( y =1;y <=3;y++) {
                for( z =1;z <=3;z++) {
                        if(x !=y && x !=z  && y !=z  )
                        { printf("%d %d %d\n",x,y,z); } } }
```

For three values first three nested 'for' loops are used to generate 3^3=27 arrangements and finally in the innermost loop a if () condition is used to select only non repeating arrangements (=3!) for printing.

But the coding logic is too restrictive in the sense that if we want to increase the number of symbols to 4 and find all distinct arrangements (=4! similarly) the program is to be rewritten entirely and recompiled. This is hazardous **in terms of coding**. Moreover, if the code is to run as part of an entire automated process then, we have no scope of doing that change in the middle.

Furthermore by this way we can arrange only 3 consecutive integers. By modifying the program slightly perhaps we can arrange any three symbols other than integers but cann't generalize it for any number of symbols. In the theory of algorithm this problem is identified as variable loop problem. Such crisis occurs in many areas of programming.
The problem can be handled in two ways: either (a) the programming logic is to be changed radically or (b) some work around may be thought by making use of the facility/feature provided in the programming language and OS.

In the first direction some people have tried to handle the matter by placing an exit condition after each loop where he checks if the number of symbols is equal to the order of depth of the current loop. But this solution is not acceptable for it cannot get itself free from fixing the max number of loops nested. The skeletal coding goes as under :

while(<some incremental condition on i>refers to first symbol>)

//increment of i refers to increment of no. of symbols

```
{  .....................
   ......................
    If(number of symbol==1)
        {if(print condition) {print symbols and
exit}
    else
    {
       while(<some incremental condition on
i->refers to second symbol>)
         { .....................
           If(number of symbol==2)
                    {if(print condition) {print
symbols and exit}
            else
           {
              ..............
             { ......................
               ......................
              { whlle(.....)
              }
        ....}
    }
```

Another similar attempt using switch case technique is found in the site of Mac Rumors [4].
The code part (switch case) goes someway like one given under:

Case 1

variables = 1
//start loop_1
for (..,...,...)

```
{
   function(variable 1) //call function(variable
1) with one variable to process e.g. permute.
 } // end loop_1
```

Case 2

```
variables = 2  // start loop_1
for ( ..,...,...)
{
   for ( ..,...,...)  //start loop_2
   {

          function(variable 1, variable 2) // call
function to process with two variables

   }  //end loop_2
  }//end loop_1
```

Case 3

```
variables = 3 // start loop_1
for ( ..,...,...)
{
 for ( ..,...,...)  //start loop_2
    {
   for ( ..,...,...)  //start loop_3
      {
                function(variable 1, variable 2,
variable 3) // call function with 3 variables
      }//end loop_3
    }//end loop_2
 }//end loop_1
.....
.....
```

Case n

```
variables = n
//start loop_1
for ( ..,...,...)  //start loop_1
    {
  for ( ..,...,...)  //start loop_2
     {
     for ( ..,...,...)  //start loop_3
       {
.....
.....
       for ( ..,...,...)  // start loop_n
        {
              function(variable 1, variable 2,
variable 3, ..., variable n)
           // call function with n variables
        } end loop_n
.....
.....
}//end loop_3
}//end loop_2
}//end loop_1
```

But all these are limited in the form of initial proposal or suggestion– there is hardly any comprehensive solution offered so far in this direction in the literature. In the first direction one significant advancement is reportedly achieved by E. Skordalakis and G. Papakonstantinou [1]. According to them a new control structure is constructed which is suitable for expressing a variable number of nested loops. This control structure is useful in combinatorial problems (explained above) as well as in equivalent problems like graph coloring etc. The rigorous discussion of their work is not the purpose of present work. Interested readers may be referred to Ledgard and Marcotty [6] and Nijenhuis and Wilf [7].

Another possibility might be deploying the recursive technique but without going into detailed logic and coding one can say that for moderately large n (>25 say) the program may fail or be very slow for excessive requirement of stack memory (hardware limitation) which has been the main drawback of recursion (vide Tenenbaum, Yedidyah Langsam [8]).

Here is performance chart in any backtracking type of algorithm:

N	number of perms	million/sec	billion/sec	trillion/sec
10	*3628800*	*Insignificant*	*insignificant*	*insignificant*
11	*39916800*	*seconds*	*insignificant*	*insignificant*
12	*479001600*	*minutes*	*seconds*	*insignificant*
13	*6227020800*	*hours*	*seconds*	*insignificant*
14	*87178291200*	*day*	*minute*	*seconds*
15	*1307674368000*	*week*	*minute*	*seconds*
16	*20922789888000*	*Months*	*hours*	*seconds*
17	*355687428096000*	*years*	*days*	*minutes*
18	*6402373705728000*	*Impossible*	*months*	*hours*
19	*121645100408832000*	*Impossible*	*years*	*months*
20	*2432902008176640000*	*Impossible*	*impossible*	*years*

[Courtesy: Prime generation method by R. Sedgewick [5] (Princeton University)]

In the second direction some naive attempts can be found on net by some armatures but they do not offer comprehensive solution e.g. using paste macro with token feature in C.

2. PROPOSED STRATEGY

In this article the authors have suggested an effective and easy to implement work around by making use of a 'code generating code' technique. Here, the input number n is treated as a parameter of the code generator program according to which the parent program generates another program containing n number of loops (determined by the value of parameter n). The child program is saved in a separate file and then compiled. The final executable program then actually produces the arrangements or similar outputs as desired– this saves the task of human intervention of rewriting and recompiling completely.

The code generator program is written in the simple C language but it generates a C++ code because the successful coding of second program (which actually generates the arrangements) relies on some features of C++ not present in C e.g., the on the fly declaration of variable data type within a loop and powerful features of cout/cin verbs for their ability to print numeric/alphabet without any specifier etc.

Now if the parent and child programs (whose name is fixed already) are kept one by one in a batch file (a.sh shell file for Unix type OS and a .BAT file in case of Windows type OS) to produce a single project file and given to user then the code generation part becomes transparent to the user. He just inputs the value n of symbols to be arranged and gets all possible arrangements. Here the program is just free from any predefined maximum number of nested loops. The maximum value of n is restricted by compiler/hardware limitation only.

Further, the batch file may be converted to an executable .exe (by using standard OS level tools like bat2exe in Windows or .sh executable in Linux type OS etc.).

e.g.
>tcc –n<PATH>\codnew.c
<PATH>\codnew.exe
>tcc -P -n<PATH>myout.cpp
<PATH>\myout.exe

2.1 Algorithm and Coding

The actual coding is given below:

Initial coder program in .c

```
#include <stdio.h>
#include <stdlib.h>
main()
{ int i,j,k,n;
  char s[2],ar[10][3]; // accommodate up to 10 variables for 10 letters
  char *t="a";
  FILE *fptr;
 if ((fptr=fopen("myout.cpp","w+"))==NULL)
    {printf("Error : cannot create file !!\n"); exit(0);}
    printf("n= "); // no of letters to arrange-one variable for each letter
   scanf("%d",&n);
  fprintf(fptr,"#include <iostream.h>\n");
  fprintf(fptr,"#include <stdio.h>\n");
  fprintf(fptr,"#include <conio.h>\n");
 fprintf(fptr,"main() {\n");
fprintf(fptr,"int cnt=1,x=1,y=1;\n");
fprintf(fptr,"int n=%d;\n",n); // taking the val of n to myout.cpp for some use
fprintf(fptr,"cirscr();\n");
for(j=1;j<=n;j++)
{ strcpy(ar[j],strcat(t,itoa(j,s,10)));
printf("ar= %3s\n",ar[j]);
strcpy(t,"a");}
for(j=1;j<=n;j++)
{fprintf(fptr,"for( int %2s =1;%2s <=%d;%2s++) {\n",ar[j],ar[j],n,ar[j]);}
 fprintf(fptr,"if(");
for(j=1;j<= n-1;j++)
  {   for(k=j+1;k<= n;k++)
      {  if(j< n-1 ) // && k < n)   { fprintf(fptr,"%2s != %2s && ",ar[j],ar[k]);  }
  else  {   fprintf(fptr,"%2s != %2s ",ar[j],ar[k]);   }
     }
}
fprintf(fptr,")\n");
```

```
fprintf(fptr,"{ gotoxy(x,y);\n");
fprintf(fptr," cout ");
for(j=1;j <= n; j++)
{ fprintf(fptr," << %2s ",ar[j]); }
fprintf(fptr,"<< \"(\"<<cnt<< \") \" << \" \";cnt++;x=x+n+5;\n");
fprintf(fptr,"if ( x > (n+5)*8 )\n");
fprintf(fptr,"{ y=y+1;x=1;}\n");
fprintf(fptr,"if ( y > 48 )\n");
fprintf(fptr,"{ getchar();clrscr();y=1;x=1;\n");
fprintf(fptr,"cout << \"press return to continue..\";\n");
fprintf(fptr,"getchar();}}\n");
for(i=1;i<=n;i++) // for loop }s
{ fprintf(fptr,"}\n");}
fprintf(fptr,"return 0;} // end main");
fclose(fptr);
return 0;
} // end of main
```

2.2 Generated .cpp program (for say, n=4 in the above program)

```
# include <iostream.h>
# include <stdio.h>
# include <conio.h>
main() {
int cnt=1,x=1,y=1;
int n=4;
clrscr();
for(int a1 =1;a1 <=4;a1++) {
for(int a2 =1;a2 <=4;a2++) {
for(int a3 =1;a3 <=4;a3++) {
for(int a4 =1;a4 <=4;a4++) {
   if(a1 != a2 && a1 != a3 && a1 != a4 && a2 != a3 && a2 != a4 && a3 != a4 )
       {gotoxy(x,y);
            cout  << a1  << a2  << a3  << a4 << "("<<cnt<< ") " << " " ;cnt++;x=x+n+5;
            if (x > (n+5)*8 )  {y=y+1;x=1;}
           if (y > 48 )
        { getchar();clrscr();y=1;x=1;
          cout << "press return to continue..";
        getchar();}}
}}}}
return 0;} // end main
```

The outputs of the last program (child) for n=4 inputs in the parent program are:
1234(1), 1243(2), 1324(3), 1342(4), 1423(5), 1432(6), 2134(7), 2143(8), 2314(9), 2341(10)
2413(11), 2431(12), 3124(13), 3142(14), 3214(15), 3241(16), 3412(17), 3421(18), 4123(19)
4132(20), 4213(21), 4231(22), 4312(23), 4321(24)

3. Conclusion

The technique used in this work is surely effective for practical purposes. The success of the program lies on creating equal number of variables (corresponding to the number of symbols) a0, a1, a2,..., an in the target program myout.cpp as well as the number of nested loops dynamically.
Authors note that in output, the number shown in the first bracket keeps track of the total number of arrangements generated (=24 as shown in the output produced).

The program can be easily generalized to generate the arrangements of any symbols than integers only (which is very tough using recursion algorithm) and generating two dimensional arrangements like matrices (total number of different nxn square matrix outputs out of n2 given symbols). Further, it can be used as a independent module and inserted to any project as a part which can exchange data in between.

Moreover, this program may be generalized to generate all arrangements of r different symbols in n different positions where n, r (r<=n) will be input. Last of all, this technique may be used to generate any number of variables in the runtime which is also a much sought for among the programmers.

The authors invite all interested persons to look into these as future scopes.

REFERENCE

[1] Skordalakis, E. and Papakonstantinou, G., A Control Structure for a Variable Number of Nested Loops, The Computer Journal, Vol. 25, No. 1, 1982, pp.48–51. https://doi.org/10.1093/comjnl/25.1.482

[2] McKenzie, B. J. and Takaoka, T., Short note: A Control Structure for a Variable Number of Nested Loops, The Computer Journal, Vol. 26, No. 3, 1983, pp. 282–283. https://doi.org/10.1093/comjnl/26.3.282

[3] Gottfried, B. S., Programming with C, Schaum's Outline Series, 2nd Edition, McGraw Hill Publication.

[4] URL of MacRumors site: http://forums.macrumors.com/archive/index.php/t-281270.html

[5] Sedgewick, R. and Wayne, K., Algorithms, 4th Edition, Princeton University

[6] Ledgard, H. F. and Marcotty, M., A Genealogy of Control Structures, Communications of the ACM 18 (No. 11), No. November, pp. 629-639, 1975.

[7] Nijenhuis, A. and Wilf, H. S., Combinatorial Algorithms, Academic Press, New York, 1975.

[8] Langsam, Y., Augenstein, M. J. and Tenenbaum, A. M., Data Structures Using C and C++ 2e, Pearson, 2006