Nibble Review Board

THE C COMPILER

Fast, transportable programs and a high level of programming flexibility are features of the C-programming language. The Nibble Review Board looks at a C-compiler for the Apple and finds a comprehensive programming system.

Review by Cornelis Bongers
Erasmus University
Postbox 1738
3000 DR Rotterdam
The Netherlands

A lot attention has been given during the past few months to the C-programming language. Byte magazine, for example, devoted an entire issue to C. Apparently, C is considered by some to be one of the "up and coming" languages.

The primary reason for the popularity of C is the high degree of transportability of C-programs. Furthermore, C is a flexible language. For example, when the execution time of certain parts of the code must be minimized, you may write this code in machine language and connect it to the C-program or use specific C constructs, such as register variables.

In this review we will discuss the Aztec C65 compiler, version V1.05C, for the Apple II and the //e. The only (extra) hardware that is needed to use the C-compiler on an Apple II is a 16K RAM card (language card). In principle, it is possible to use the C-compiler on a one-drive system, but since the C-compiler is heavily disk-based, this will require some "disk-shuffling" and copy operations.

Product Description

The title on the manual says simply "C-compiler". I would, however, rather say that the package is a complete C-system. The compiler is in fact only one vital part of the system.

An essential point is that Aztec C supports the full Kernighan and Ritchie (see the References section of this article) C-language standard, except for bit fields. This means that generally no modifications will be needed when implementing a standard C-program on your Apple. (Not all implementations of C share this feature.)

NIBBLE REVIEW CARD

AZTEC C65 COMPILER
Manx Software Systems
Box 55, Shrewsbury, NJ 07701
$199.00

CATEGORY	RATING
Performance	4
User Friendliness	3
Documentation	2
Overall Rating	4

LEGEND: Five Apples = Excellent
Four Apples = Very Good
Three Apples = Good
Two Apples = Fair
One Apple = Poor

The complete package comes on three reversible diskettes, which contain a total of nearly 650K of code. The C-system disks are not copy-protected, so back-up copies can be made freely. However, on an accompanying sheet called the End User License Agreement, the user is explicitly informed about possible consequences of making copies for others. What was new to me is that users who have two Apples at different places (or in my case a BASIS 108 and an Apple), and who use the C-system on both machines are required to buy two separate licenses. In my opinion this pushes things a bit too far.

The C-system files and programs on the diskettes can roughly be categorized as follows:

1. The SHELL command interpreter
2. The VED screen editor
3. The compilers
4. The assemblers
5. The linker
6. The libraries
7. The archives
8. The utilities

Rather than giving a formal description of the system software, I will use a sample C-program as a guideline to discuss the list above.

Let's suppose that we want to make an executable object file from the program in **Listing 1**. This program generates the "n!" permutations of the numbers 1 to n in increasing order. Thus, when the number 3 is input, the result is:

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

As can be seen from **Listing 1**, the C-source consists of a main program and the function "permutate." In the main program, the user is requested to enter a number. The call to the function "gets" puts the number in ASCII format in the character array "buf". The number string is then translated to an integer by means of the call to "atoi" and the result of this conversion is passed to the function "permutate."

The first thing that is done in "permutate" is filling the array "perm" with the numbers 0 to n. Note the peculiar form of the for statement. The first expression in this statement (i=0) initializes the loop variable. If the second expression (i<=n) evaluates to "true" (i.e. a nonzero result), the loop is executed another time. The third expression (i++, which stands for i=i+1) is evaluated after each execution of the loop.

There is a lot of freedom regarding the specification of the loop parameters. For example, you may substitute more expressions, separated by commas, for each of the three expressions mentioned above. The last loop in the program provides an example.

The body of a loop consists of either a single statement or a group of statements (block), enclosed by braces. With this information at hand, it should be no problem to find out how the program works. This is left as an exercise to the interested reader.

To convert the program to an executable object, the C-system has to be powered up. Before this can be done, a standard DOS 3.3

master must be booted, for there is no DOS on the C-system disks.

Next the SHELL can be installed with the command **BRUN SHELL**. The SHELL replaces the command interpreter part of DOS 3.3 and takes care of the interaction between the user and the low-level I/O routines. This implies that most of the familiar DOS commands, such as CATALOG, BRUN, etc. are replaced by others. The disk file format is not changed, so files created with the C-system can be processed by FID or other DOS 3.3 programs.

The SHELL supports the following commands:

boot	- Reboot the system.
bye	- Exit to the Apple monitor.
call	- Similar to the BASIC CALL instruction. (Hex addresses can be specified though.)
cat	- List files on the screen or send them to another device. (For example, to print the file marjo.c you can enter: **cat marjo.c >pr:**.)
cd/ce	- Change the data/execution drive.
cp	- Copy files.
load/run	- Load and execute a program.
(un)lock	- Same as DOS 3.3 LOCK/UNLOCK.
maxfiles	- Same as DOS 3.3 MAXFILES.
ls	- Same as DOS 3.3 CATALOG.
mv	- Same as cp, except that the source file is deleted.
rm	- Delete a file.
save	- Similar to DOS 3.3 BSAVE.

It takes some time to get accustomed to the new commands, but when you get the feel of them, they work conveniently. The cd/ce commands in particular are very handy. With cd you can change the number or slot of the data drive. This has the effect that by default, all data files will be loaded from or saved on the disk in that drive.

You can also change the number and slot of the execution drive with the ce command. The SHELL will then look for (nondata) files on the execution drive if it doesn't find them on the data drive. This feature allows the user to distribute utilities, compilers and libraries over two disks, without the need to remember where a file is stored.

The load/run commands are convenient for those who have a one-drive system. With "load" you can load a system (or other) program from one disk; next, you may change disks so that the disk with the files needed by the system program is in the drive; and then you can type **run**, followed by a parameter list. The run command executes the most recently loaded program and the parameter list is passed to this program. This construct works for most system programs, except for (among others) the compiler.

"Aztec C supports the full Kernighan and Ritchie C-language standard, except for bit fields."

Overall, the SHELL provides an excellent environment for the C-programmer. Most parts of the SHELL are written in C, which is in itself a recommendation for the Aztec C-system. The only noticeable effect is that the system responds a fraction slower than DOS 3.3.

From SHELL command mode, programs can be executed by entering the **filename** <**RETURN**>. On typing **VED**, the screen editor is invoked. VED is also written in C and performs reasonably well. To enter a program, you must press the 'I' key, which puts you in insert mode.

On pressing <**ESC**>, you go back to edit mode, and in this mode corrections can be made. Single characters or whole lines can be deleted or replaced. There are also commands to go to the next or previous character/line, to the start or end of the line, to the top or bottom of the screen, and to the nth line of the file. User-specified strings can be searched (but not "automatically" replaced) and it is possible to move text around.

It takes some time to learn the editor commands, for the command mnemonics bear no relationship to any other editor I have worked with. For example, the $ symbol is used to move the cursor to the end of a line. Fortunately, this needn't worry the potential buyer of the C-system very much, for the source of VED is supplied on one of the disks, so that it can easily be customized.

VED is not bug free, and this causes an occasional system hang-up. For example, when you start with a new file and the first thing you do is move the cursor to the middle of the screen (by pressing **M**), you usually get some pretty weird results. Furthermore, VED can only handle text files but doesn't check on the file type of the file you want to edit. So, when you accidentally read in a binary file, there is a good chance that you will have to reboot to get things going again.

Overall, I would not consider these "bugs" to be very serious though, for I noticed them only when testing VED (i.e. searching for bugs), and they did not bother me when I was entering or editing text normally.

After the program has been entered and mistakes have been corrected, it is saved on disk and we return to the SHELL. The next steps are compiling and linking the program. During the compilation step, the source code is converted to a relocatable file. Since the program contains calls to functions that are not part of the program (i.e. gets, atoi and printf), these functions must also be loaded and connected. This is done in the link step. During this step, the C-system library is searched for unresolved references.

Assuming that we name the permutation program permutate.c, it could be compiled by specifying:

c65 permutate.c

The compiler creates a temporary source file ($TMP.$$$) with 6502 assembler instructions. Next, the compiler invokes a 6502 assembler, which assembles $TMP.$$$ and creates a relocatable object file named permutate.rel. The assembler normally deletes $TMP.$$$, but if you want to look at the assembler source, the compiler can be forced to stop after the compilation step. You can then modify the assembler source and assemble it in a separate step. The assembler can thus also be used "stand-alone."

The compiler can recognize 99 different errors and when an error is detected, the line number of the line (in the source) where the error occurred and the error number are displayed. The error handling of the compiler is not foolproof though, for the compiler will hang on some errors. For example, when you forget the parentheses around the expression after "while" or "if," you will get lots of error messages and after that the system will die.

I could always recover from this situation by pressing <**RESET**>, but there is of course no guarantee that all vital parts of the operating system are still intact after the system hang-up. The C verifier "lint" is not supplied with the Aztec system, so there is no easy way to prevent the problems mentioned above from occurring. It must be stressed though that, as

continued on next page

The Aztec C Compiler (Cont.)

long as a C-program was error-free, the compiler worked flawlessly in compiling all of the programs I tried.

Finally, the linker must be invoked to load and link the "gets," "atoi" and "printf" functions. The function "atoi" belongs to the class of utility functions, many of which are supplied in the different libraries. The functions "gets" and "printf" on the other hand belong to the standard I/O functions. This class provides the link between the C-language and the specific system that is used.

I/O operations are not explicitly defined in C, so the user is strongly dependent on the functions supplied by the distributor of his C-implementation. Manx Software Systems did a good job here. All the functions you need and many more are present (including sources).

The linker is invoked with:

ln permutate.rel sh65.lib

This command instructs the linker to search the library sh65, which contains the standard I/O and utility functions, for the functions needed by permutate. The output of the linker is an executable object file named permutate. Permutate can be executed from SHELL command level, just like any other C-program, by entering: **permutate** <RETURN>.

An important point is that the permutate object we created above can only be executed under the SHELL. This is because during the link step only those functions will be loaded that are not yet present in the SHELL. If a function is already present, the linker substitutes only the address of this function in the object. It will be clear that this usually reduces the size of the object considerably, but prohibits execution of the object in a non-SHELL environment.

It is, however, also possible to develop C-software for use under DOS 3.3 or another operating system. The only thing you have to do is use a different library, namely the sa65 library, instead of the sh65 library in the link command. The sa65 library contains all the functions that are normally present in the SHELL, so an object program constructed with this library can be executed under DOS 3.3.

Apart from the standard system I/O and utility library, there is also a library with a complete set of mathematical functions. All of these give double precision results (15-16 significant digits). The floating-point exceptions: overflow, underflow and division by zero are supported and can be trapped by the user program.

A problem associated with the use of "heavy" software on microcomputers is that object files may become so large that they do not fit in memory anymore. Aztec C offers two solutions to this problem.

The first solution is very elegant. It consists of the support of a pseudo-code compiler, named cci. The cci compiler produces a pseudo-code assembly source file that is

subsequently assembled by a pseudo-code assembler. The relocatable pseudo-code object file, generated by the assembler, can be processed by the linker to generate an executable object file. Such an object file will generally be smaller (the manual says usually by more than 50%) than the corresponding 6502 object file. The price you have to pay is in terms of execution time, for a pseudo-code object will execute more slowly than a 6502 object by a factor of from 5 to 20.

A point worth noting is that relocatable pseudo-code objects and relocatable 6502 objects can "simultaneously" be processed by the linker. This means that you can compile time-critical routines with c65 and the others with cci. The linker will then create a mixed pseudo-code/6502 object, which behaves just like a pure 6502 object.

The second solution to reduce the size of objects is to use overlays. With overlays, the program is divided into several segments. There must be one root segment which always remains in memory. The other segments are loaded only when there are needed. It is possible to "nest" overlays; thus a segment that is an overlay itself may load other overlays.

There are many other interesting options and features of the C-system that would be worth mentioning. However, space limitations prohibit further explanations, so the remaining highlights are briefly summarized below.

1. Archives containing more than 150 source files are included. Most of the standard I/O, system I/O and utility routines mentioned in the Kernighan and Ritchie book are available. There are also several preprepared "include" files — among others, STDIO.H.

2. There are utilities to compare files, to dump files (hex/ASCII), and to print sorted Symbol Tables.

3. With the CONFIG program, the SHELL Device Driver Tables can be adapted to your system configuration.

4. The source code of the device drivers is included and custom drivers can be added and loaded with a separate program.

5. The SHELL supports batch processing; that is, you can create a text file with SHELL commands and execute it. Parameters can easily be passed to the commands and a loop facility is implemented.

6. The SHELL also supports input/output redirection (but no pipelining).

7. You may create your own libraries (that can be used by the linker) with the MKLIB program.

8. Interfacing C-programs with assembly language programs is fully supported and the calling conventions are explained.

9. It is possible to develop ROMable code, although there are some restrictions. You must, for example, (probably) rewrite some start-up routines.

Configuration

The C-system was tested on an Apple II Plus with a nonstandard 80-column board and on a BASIS 108. (The C-system also works on 40-column Apples; upper-case is then displayed in inverse.) Configuring the system to 80-columns was no problem. But since the 80-column card was not standard, some extra work had to be done in filling in the configuration parameters of the Control Code Table. However, if you have an Apple //e with the standard 80-column card or the Videx Videoterm or Smarterm card, configuration can be done in less than a minute.

Configuring the C-system for the 80-column BASIS 108 turned out to be slightly more difficult, but since the manual gives detailed information about device drivers and Device

continued on next page

The Aztec C Compiler (Cont.)

Table entries, it still was a relatively easy operation.

The C-system worked immediately with my Itoh printer, but two line feeds were given after each carriage return. This problem was easily solved, however, by reconfiguring the printer flag with the CONFIG program.

Performance

To test the performance of the C-compiler, I entered some of the test programs described in "Five C Compilers for CP/M 80." (See the References section.) The tests in that article (reference 2) were done on a 2-MHz Z80 system, so the comparisons below are of importance to Z80 card owners, since they also have the option to buy a CP/M version of C. The results of the Aztec C-compiler are listed in **Table 1**.

TABLE 1: Performance of the Aztec C-Compiler[1]

Program	Execution Time in Sec.	Compile & Link Time in Sec.	Size of Object in K
Primes	37	210	9
Fibonacci	739	175	5.5
String-length	123	180	5.5
String-length-r	54	180	5.5

1. Benchmark programs taken from Byte Magazine, August 1983, pp. 110-130.

The primes program uses the "sieve of Eratosthenes" algorithm to compute all primes between 1 and 16380, and tests the compiler's ability to handle loops and arrays.

The fibonacci program tests the overhead involved in making function calls by recursively computing the 24th term of the fibonacci series. The primes program as well as the fibonacci program were executed 10 times in a loop, so only 3.7 seconds were needed by the primes program to find the 1899 primes below 16380.

The third program, string-length, indicates the efficiency of string (pointer) handling, by passing a string (pointer) to a function, which computes the length of the string 25,000 times.

continued on page 131

The Aztec C Compiler (Cont.)

The string-length-r program is the same as string-length, except that the character count variable was declared to be of storage class "register."

When the test results are compared with those published in reference 2, it turns out that the sizes of the generated objects, as well as the compile and link times, do not deviate much from the overall averages (per program) realized by the other compilers. The deviations that do exist are all in favor of the Aztec C-compiler, except for the size of the primes program, which was somewhat larger than average.

It should be noted that all of the programs in **Table 1** are relatively small. (All source files were less than 1K.) The sizes of the corresponding objects then become relatively large, due to the fixed overhead of I/O routines. However, when compiling a large program, the relative difference between source and object will become much smaller. For example, when you compile VED, which consists of about 17.5K of source code, the resulting object is only 15K. By the way, compiling and linking VED does take quite a long time (more than 40 minutes).

Regarding the execution times of the primes and the two string-length programs, the Aztec C/6502 combination outperformed all of the compilers discussed in reference 2. More precisely, relative to the fastest corresponding programs in reference 2, the primes program ran 38% faster, the string-length program 11% faster, and the string-length-r program 25% faster.

On the other hand, the fibonacci program realized an extremely long execution time and it ran more than three times slower than the slowest program tested in reference 2.

Incidentally, the latter was also compiled with an Aztec (CP/M) compiler. I could not resist the temptation to look "under the hood," and it turned out that there is indeed a subroutine call to a fairly large subroutine (.csav#) at the start of each function, which adds a considerable amount of overhead. Furthermore, function calls themselves are placed between stack-save and stack-restore routines, which also consume extra time.

Since the 6502 maintains only a 256-byte stack, which is too small for C-programs, a pseudo-stack is set up, and at each function call the relevant parameters are pushed on this pseudo-stack. The existence of a pseudo-stack explains largely why function calling with the 6502 goes much slower than with the Z80. Specifically, the latter processor is better equipped for stack operations, as it has a 16-bit stack pointer.

I also tested the performance of the permutation program, listed in **Listing 1**. This program was compiled with the c65 compiler and the cci compiler, and both versions were linked with the different libraries. The results are displayed in **Table 2**.

As can be seen, the increase in execution time is about a factor of 6 when the cci compiler is used instead of the c65 compiler. When comparing the size of the object generated by the c65/sh65 combination with that of the cci/shint combination, it appears that there is a reduction of more than 50%. (This agrees with what the manual says.) However, the size of the object generated by the c65/shint combination differs only by a small amount from that of the cci/shint combination. Since the c65/shint combination generates code that runs much faster, this seems to be the best choice here.

The reason why the c65/shint combination turns out to be the best is that the generation of the permutations is done with 6502 instructions, and the I/O operations (loaded from the shint library) are done with pseudo-code instructions. Since the permutation program uses little I/O, it doesn't matter that slow pseudo code is used for I/O.

As far as execution time of the permutation program is concerned, the c65 compiler performs better than most other compilers I am familiar with. The MBASIC compiler produced faster code though. The compiled

continued on page 133

The Aztec C Compiler (Cont.)

MBASIC version of the permutation program needed only 33 seconds to generate the 8! permutations (on a MicroSoft Z80 card). This implies a speed advantage of a factor of 2 over the C-version. So, I rewrote the C-permutation program to speed it up a bit. The result is displayed in **Listing 2**.

As can be seen, all time-critical operations are now done with pointers. Pointer handling is very efficiently done by Aztec C, for the execution time (with the print statements removed) dropped to 12 seconds. This is exceptionally fast, particularly when compared to Applesoft, which needs nearly three quarters of an hour to get the job done.

A final (small) test concerns the speed of floating-point calculations. C supports only double precision floating-point calculations, which results in high precision, relative to other languages, but in longer execution time. In Aztec C, 1000 multiplications of floating-point numbers take 18.2 seconds. Applesoft accomplishes the same thing in 4.3 seconds.

Correcting for the difference in precision (Applesoft uses a 4-byte mantissa while C uses a 7-byte mantissa) by assuming that execution time is a quadratic function of the number of mantissa bytes, we find that a multiplication in C takes 40% more time ("per digit") than in Applesoft. When evaluating mathematical functions, however, this percentage increases. For example, C needs 170 seconds to compute 1000 sine functions, while Applesoft needs only 27 seconds.

User Friendliness

Judging the user friendliness of a software package is a rather subjective matter. Personally, I rate the degree of user friendliness of the Aztec C-system somewhere between medium and high.

The SHELL environment is about as user friendly as the DOS 3.3 environment. Configuring the C-system and processing C-programs (i.e. editing, compiling, linking and running) is not difficult, but you have to spend some time in learning the commands and options. As already mentioned, error handling is not always done correctly. The compiler, in particular, is sensitive to user errors, so in this regard there is room for improvement.

I experienced several system hang-ups, but none of these caused loss of data. I want to stress though that most system hang-ups were caused by run-time errors in the C-programs I wrote, and the C-system can of course not be blamed for such errors.

It is a fact that the C-language is not a very "easy" language and C-programs are harder to debug than Applesoft (or Pascal) programs. For example, if you exceed an array bound, no one will tell you and the C-programmer is on his own in finding out what caused the resulting mess. C simply assumes that a program contains no bugs. This implies that C and the C-system may be less suited for those who are just starting out with computers. On the other hand, those who have a reasonable amount of experience with the Apple should encounter no real problems in working with C on the Aztec C-system.

Documentation

The documentation of the C-system (about 140 pages) is reasonable. It does not describe the C-language, so if you have no experience with C you have to buy a separate text. A good choice, also recommended by the manual, is *The C Programming Language*. (See the References section.) This book provides a C-tutorial, explanations of all the C-constructs and many examples. Furthermore, the book integrates well with the C-system, since many utilities in the book are supplied with the system.

Explanations in the documentation are brief, which means that you must read things carefully. For example, I had some problems in getting the ARCH program running, but after closer reading, it turned out that all the necessary information was there.

The documentation doesn't mention everything though. For example, the n-option (search next occurrence of string) is mentioned on VED's Help screen, but not in the documentation. Furthermore, VED's wq option (save file and quit) is mentioned neither on the Help screen nor in the documentation.

A more serious omission concerns an explanation of the error messages of the linker. For example, when the program "sinf" uses

continued on next page

The Aztec C Compiler (Cont.)

the "sin" function, the library of floating-point routines (flt65) must be specified. However, when you enter **ln sinf.rel sh65.lib flt65.lib**, you will get mysterious error messages. These don't indicate that only the sequence of library specifications is wrong. (The right specification is **ln sinf.rel flt65.lib sh65.lib**.)

The difficulty level of the documentation roughly corresponds to the difficulty level of the Kernighan and Ritchie text.

There is a fairly complicated technical section, but you need only read it when you want to do some of the more advanced applications, or when you want to use the C-system with nonstandard hardware. The manual has a table of contents, but no index and glossary. Most commands and functions are illustrated with examples and a good tutorial is included.

There is a possibility of getting regular updates of the C-system by paying an annual maintenance fee of $50 (plus $35 if you happen to live outside the USA). A new release can also be purchased separately for $30. There is no replacement policy, which is not unreasonable since the C-system is not copy-protected.

"The documentation doesn't mention everything..."

Conclusion

The Aztec C-system is one of the finest software packages I have seen. The package is particularly valuable because of the many C-sources that are included. This gives the beginning C-student the opportunity to study C-programs written by an expert, which is a good way to get the hang of a language. Furthermore, the fact that Aztec C fully supports the Kernighan and Ritchie standard is important, for in this way you need not waste your time in studying implementation differences. It also considerably increases the probability that the C-programs you develop on the Apple will run without modifications on other systems.

There are definitely bugs in the Aztec C-system, but none of these appears to be very serious as they do not prohibit normal operations (as long as the user does not make too many errors). In fact, the number of bugs is surprisingly small, taking into consideration the fact that we are talking about some 650K of code. Nevertheless, bugs should in principle not occur and Manx Software Systems should attempt to remove them in the next release.

The 6502 code generated by the sh65/sa65 compilers is generally very fast, except for the code generated at function calls and floating-point operations. Compile and link times are reasonable and the same applies to the sizes of the generated 6502 objects. When the size of the object is a critical factor, a considerable size reduction can be realized by using the

pseudo-code compiler and/or the pseudo-coded libraries.

Again, the Aztec C-system may be less suited for those who do not have much experience with computers. Programming in a compiler-linker based language is quite different from programming in an interpreter ased language. It requires, among other things, much more attention and patience from the user. This applies not only to writing programs but also to typing in programs. Repairing one single syntax error, for example, may cost you a considerable amount of time. Run-time errors are even worse, since you have to track those down by studying the source, without the aid of run time error messages.

Summarizing, I can recommend the Aztec C-system to anyone with a medium to high degree of experience with the Apple, who is interested in learning C, and/or wants to develop C-programs in a "UNIX-like" environment.

For the time being, a few bugs have to be taken for granted, but in view of the overall high quality of the C-system, these will undoubtedly be corrected in the near future.

Author's Note: My thanks to William Schouten for his many valuable comments and suggestions.

References

1. *The C Programming Language*, B.W. Kernighan and D. Ritchie, Prentice-Hall, Inc., New Jersey, 1978.

2. "Five C Compilers for CP/M 80," C.O. Kern, *Byte Magazine*, August 1983, p. 110-130.

LISTING 1

```
/* Permutation generator */

#define BUFFER 10

main ()

{
    char buf[BUFFER];

    printf("Enter number: ");
    permutate(atoi(gets(buf)));
}

#define MAX 20

permutate(n)
int n;

{
    static int perm[MAX];
    register i,j,t,k,count=0;

    for (i=0; i<=n; i++)
        perm[i]=i; /* init permarray */

    while (1) < /* force continuous loop */

        count++;

        for (j=1;j<=n;j++)
            printf("%d  ",perm[j]);
        printf("\n");

        for (i=n-1;perm[i]>=perm[i+1];i--)
            ;

        if (!i)
            break; /* stop if i=0 */

        for (j=n;perm[j]<=perm[i];j--)
            ;

        t=perm[i];
        perm[i]=perm[j];
        perm[j]=t;

        for (j=n,k=i+1;k<j;j--,k++) {
            t=perm[k];
            perm[k]=perm[j];
            perm[j]=t;
        }
    }
    printf("No of permutations is %u \n",count);
}
```

LISTING 2

```
/* Permutation generator, pointer version */

#define BUFFER 10

main ()

{
    char buf[BUFFER];

    printf("Enter number: ");
    permutate(atoi(gets(buf)));
}

#define MAX 20

permutate(n)
int n;

{
    static   char perm[MAX];
    register count=0;
    register char i,t;
    register char *pn,*pl,*ph;

    for (i=0; i<=n; i++)
        perm[i]=i; /* init permarray */

    pn=perm+n;

    for (;;) {  /* force continuous loop */

        count++;

        for (pl=pn-1;*pl>=*(pl+1);pl--)
            ;

        if (!*pl)
            break;

        for (ph=pn+1;*--ph<=*pl;)
            ;

        t=*pl;
        *pl=*ph;
        *ph=t;

        for (ph=pn+1;++pl<--ph;) {
            t=*pl;
            *pl=*ph;
            *ph=t;
        }
    }
    printf("No of permutations is %u \n",count);
}
```