# Learning C with the SAS/C® Compiler, Student Edition

Gary Merrill, SAS Institute Inc., Cary, NC
Dee Stribling, SAS Institute Inc., Cary, NC

## ABSTRACT

By design, the C language is a flexible and powerful tool for software development. C encourages modular programming and efficient, portable applications. This paper provides introductory background on compilers, the C language, and how a task-oriented tutorial approach to learning C was developed by combining compiler features with C language instruction in a PC environment. This paper also includes a discussion of how to build and run a C program using the SAS/C® Compiler, Student Edition (also referred to as the Student Compiler). This example illustrates how the tutorial can be used with the Student Compiler to provide an easy and efficient way to learn basic principles of C program development.

## WHY SHOULD YOU USE THE C LANGUAGE?

The C programming language is one of a number of general-purpose programming languages in the group that includes BASIC, FORTRAN, COBOL, PASCAL, and PL/I. Although in some ways similar, C stands apart from these languages because C possesses certain distinctive features that recommend its use in an ever widening realm of applications. C is not a "big" language; it has relatively few keywords, only a handful of fundamental data types, and a terse but straightforward syntax. (In fact, the ANSI Standard guidelines for the C language require that the language be kept small and simple, provide only one way to do things, and be efficient and fast.) Nonetheless, C continues to grow in popularity because it combines many typical features of high-level languages with low-level features characteristic of assembly languages.

Because the high-level features of C give it the same facility as other high-level, general-purpose languages while its low-level features allow it to be used for purposes previously requiring assembly language, C has become the language of choice on many microcomputers and minicomputers. For similar reasons, C recently has made significant inroads in both applications and systems programming on mainframes as well.

Another advantage of C (particularly in light of the recent ANSI Standard for the language and IBM®'s emerging Systems Application Architecture for C) is that it is easy to write *portable* programs in C —where portable means that exactly the same source code that works on one machine will work on another. The C Language's potential for portability, coupled with the fact that virtually all machines and operating systems now boast an available C compiler, has encouraged software vendors to code new applications in C and to recode older applications in C even if this requires substantial effort. The long-term savings in having portable code that is easy to maintain is well worth the conversion effort. The SAS® System recently has undergone (and to some degree is still undergoing) such a conversion.

Finally, adding to the utility of C on the mainframe is the fact that quality compilers such as the SAS/C mainframe compiler permit *interlanguage communication*; that is, routines written in C can call or be called by other routines written in FORTRAN, COBOL, PL/I, or assembly language.

Overall, C is a well established general-purpose programming language that can be used in a wide variety of both large and small applications. The following are some typical uses for C in today's programming environment:

Operating systems
: AT&T Bell Laboratories UNIX® is written in C, as are a variety of other operating systems.

Text processors
: A number of commonly available text processors are written in C. Lattice® 's HighStyle™ desktop publishing package is a recent example.

Compilers
: The Lattice and SAS/C compilers are written in C, as are a number of compilers marketed by Microsoft® Borland, and other vendors.

Real-time laboratory or industrial data analysis systems
: Because of the ability to generate efficient machine code from C source code, C is frequently used for real-time applications on small laboratory machines and even robots. C has also been used extensively for the construction of telephone switching systems.

Embedded systems
: C is often used in programming ROMs (read-only memory areas) found in special processors such as device controllers, printers, scientific instruments, electronic scales, and home appliances.

Business and financial applications
: In recent years, C has been taken to heart by the financial community for use in a variety of business applications, including financial applications such as securities trading.

An accurate motto for C applications would be "no job too large, no job too small." The language can be used efficiently for such small jobs as device drivers as well as such large jobs as the complex and comprehensive SAS System statistical analysis package in which text processing, graphics, and operations research capabilities are all programmed using C.

In addition to being a relatively simple language to learn, the C language has the following features:

- a rich standard library of functions (routines) to handle such chores as string processing, input/output, and mathematical functions
- the ability to construct complex data structures such as lists, trees, and graphs by using standard library functions for the dynamic allocation and deallocation of memory
- the ease with which efficient code can be written using high-level language constructs
- the possibility of writing portable code that is usable across a number of machines and operating systems.

## WHAT YOU NEED TO PROGRAM IN THE C LANGUAGE

To get started programming in C, you need only a computer system that provides you with a C compiler. Since C is now available on virtually all computer systems, this initial requirement is met easily. If you have access to an IBM PC or PC clone capable of running MS-DOS® a good approach to learning C is through the SAS/C Compiler, Student Edition and its tutorial. But before going further, let's look at what a compiler is and how it enters into the programming process.

To begin, you write your program in the C language. But what is a program? A simple definition of *program* is

> a series of steps (or operations) intended to solve a certain problem or to produce certain results.

When people talk of writing a computer program, they have in mind a problem they need to solve or some task they want the computer to perform. The program, in this sense, tells the machine what to do.

A *source program* is a description of what you want the computer to do to solve the problem or perform the task. Your source code is written in C. Once you have written your source program, you want the computer to execute or *run* it. This requires that the computer understand your program. But the computer, which in reality is just a complicated arrangement of very small switches and circuits, cannot understand your program when it is in its source form (C code). The program that the computer can run is actually just a particular arrangement of these switches and circuits. This version of the program is called the *machine program* or the *executable program*.

You need to change the source program (that you can read and understand) into the machine program (that the computer can understand and execute). The way to do this is to use another program called a *compiler*. The compiler is really a translator in the sense that it translates your source program into a machine program as illustrated in **Figure 1**.

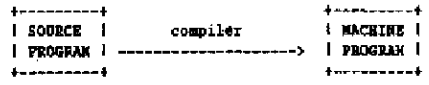


**Figure 1** Compiling a Program

Compilers are machine-specific; that is, they are designed to run on a certain kind of computer because each kind of computer has its own machine language. The Student Compiler can produce a machine program on an IBM PC (or one of its compatibly designed computers) that corresponds to your C language source program.

The following are the basic steps in creating your program in C:

1. Write the C source code for your program. (Create a file or data set containing this source code.)
2. Use a C compiler to translate the C source code into the machine program. (With some compilers, other utilities such as a link editor may be needed as well.)
3. Run the resulting machine program on your computer.

This process is greatly simplified by using an integrated product such as the Student Compiler.

## WHAT IS THE SAS/C COMPILER, STUDENT EDITION?

The Student Compiler is an easy-to-use, integrated environment that provides the ability to create and edit a file of C source code, compile that file, and run the resulting machine program. The Student Compiler also provides a debugger that allows you to step through your program in a controlled way to check it for bugs (errors in your program logic). The debugger can also serve as a learning tool. By displaying source code and variable contents, you can see exactly what is happening with each C statement you have written.

By providing an integrated *push button* environment using menus and function keys, the Student Compiler insulates you from many of the complexities encountered in using a production compiler. It allows you to edit, compile, run, and debug your program without getting involved in the details of operating system commands, compiler options, library selection, and additional utilities. By using the Student Compiler and its associated tutorial, you create a C program easily and run that program within a few moments, while a comparable attempt with a production compiler on any operating system might require some frustrating time spent searching manuals in order to familiarize yourself with various operating system commands.

Our experience with in-house training shows that the Student Compiler can be used either as a self-teaching aid or as the primary compiler in an organized course. Students using the Student Compiler typically progress faster and with a fuller understanding than their classmates who have to struggle with the complexities of an operating system and its interface with the native compiler for a given machine. With the Student Compiler, all of your time can be devoted to learning the C language rather than learning features of the operating system that are only vaguely related to compiler use.

## WHY SAS INSTITUTE DEVELOPED A C LANGUAGE TUTORIAL

Because C is increasingly becoming the language of choice for so many applications, the tutorial accompanying the SAS/C Compiler, Student Edition was created in response to the need for a simple yet comprehensive approach to learning the C language.

Although C is meant to be a simple and straightforward language, in some contexts, C looks highly technical and can appear difficult to learn. Even if the code is well commented, if you don't have the appropriate background, some C program logic can still be confusing (for example C numbers the first element of an array as 0 making the tenth element number 9). If you dive into the world of C without being aware of the basics in structure and syntax, you may be in for a frustrating programming experience. However, with a good understanding of C basics and how C differs from other common languages, you will realize that the benefits gained from writing applications in C outweigh any initial worries about learning the language.

In fact, because of C's structure and flexibility, you will find it easy to write *considerate* programs (modular and well-commented) after a good introduction to C. And, in considering the user, the C language's full-screen and graphics functions make it easy for you to write friendly interfaces and prototypes for systems, even if portions of the system are developed in assembler or some other language.

Together, the tutorial and Student Compiler provide an opportunity for you to become acquainted with C quickly and in a friendly environment. When learning to write the C programs in the tutorial, you can take advantage of the many editing and template features of the Student Compiler. And, by studying the examples in the tutorial while actually running and debugging them with the compiler, you can begin to get a feel for the simple and efficient structure of the C language.

## DEVELOPING THE C TUTORIAL

The *Student Edition* tutorial was written to give basic and practical information about the C language. Guiding principles included

- conveying important C principles and concepts
- presenting these concepts in clear and ordered steps
- organizing the concepts around specific tasks.

Since the tutorial accompanying the Student Compiler was planned with the principles above in mind, important topics are presented by level of difficulty and "need to know." Special consideration was given to areas where people new to the C language often encounter difficulty. For example, particular attention was given to concepts such as functions, pointers, and storage class, as well as the syntax concerning arrays, macros, loops, and expressions.

Since special attention was given to content, topic order, and presentation in terms of both the requirements of the C language and the different programming needs of the audience, the tutorial allows you to progress at your own speed, according to your own particular C programming needs. This means that persons learning C as a first language can learn quickly to write simple programs, while someone more familiar with programming techniques can go ahead and use the material in practical applications.

## TUTORIAL ORGANIZATION

A complete introduction to the C language is provided in Chapters 1 through 7 of the tutorial in the *SAS/C Compiler, Student Edition*.* These topics are then expanded in the last four chapters. Again, this means that the material in the book can be presented purely in terms of C basics or, by including the last chapters, as a more comprehensive introduction to the C language.

Topics covered in the initial chapters cover the fundamental aspects of most C features. These topics are

- introduction: an overview of C and the tutorial
- data: number representation and variables in C
- operators: expressions in C
- program flow: statements that provide program control
- input and output: simple ways to read and write data
- functions: how to write units of C program code
- arrays: how to handle groups of data or variables
- pointers: how to use addresses to manipulate data.

Chapters in the advanced section expand on the material presented earlier and provide a more complete background in C programming techniques. These chapters cover the following topics:

- more on operators and data types
- more on expressions and the C preprocessor
- working with data: files, structures, and storage
- introduction to data structures
- helpful hints and common mistakes.

Each tutorial learning module focuses on a particular aspect of the C language. In each module, you develop the C language skills to complete specific tasks. Each task is incorporated into a C program that can be written and tested using the Student Compiler. **Table 1** shows how tutorial modules are integrated with the examples available on diskette with the compiler.

**Table 1** Tutorial Topics and Program Tasks

| Module Topic | Task Purpose | Program Example |
|---|---|---|
| Basic structure of a C program | Using the main() function | Print a simple list |
| Data types | Describing data to program | Print sizes of types |
| Operators | Learning to use expressions | Basic calculator |
| Input/Output | Reading and writing data | Selection and display of menu items |
| Function structure | Learning to write C functions | Game using several short functions |
| Arrays and pointers | Learning about addressing storage and using pointers | Convert units of measure via tables |
| C preprocessor | Using the preprocessor | Translate language phrases |
| Structures | Working with files and storage | Simple database of names and addresses |

The next section of this paper discusses the tutorial's approach to teaching the C language and shows by example how the compiler and tutorial work together. The example introduces the C language and gives you an idea of what it is like to learn C using the SAS/C Student Edition compiler and tutorial.

## A TASK-ORIENTED APPROACH TO C PROGRAM DEVELOPMENT

C programs are composed of modules of discrete functional units. This makes it easy to use a task-oriented paradigm for C program development. The following steps outline this approach to programming:

1. Decide on principle program tasks.
2. Determine flow of control.
3. Write pseudocode for the overall flow.
4. Determine C functions for major tasks.
5. Break high-level functions into smaller tasks.
6. Code, test, and debug.

This task-oriented approach is applied in the following example, which is representative of a typical tutorial module.**

## SAMPLE TUTORIAL MODULE: INTRODUCTION

This example involves writing the C code to add, subtract, multiply, or divide two integer numbers. In other words, you will build a simple calculator program. The C tasks involved include asking the user to enter the numbers and operator, performing the calculations involved based on the value of the operator, and displaying the results. In order to do this, this module covers the basic structure of a C program and enough about statements for choices and expressions to code the tasks mentioned above. Important concepts are

- understanding the structure of C programs
- writing simple statements
- using C functions
- controlling program flow.

Keywords you need to note are

```
main()          declaration
printf()        function
switch
#include
```

A C program is composed of one or more functions, similar in concept to subroutines in other languages. A C program always begins with the `main()` or controlling function. (Functions are written with parentheses to indicate the possibility of parameters or function arguments.) You can think of a C function as a self-contained set of code that performs a specific task.

Most C programming tasks usually require functions in addition to `main()`. You can use other prewritten C functions that come with your compiler in function libraries, or you can write your own functions. You write your own functions by using C statements telling the C compiler exactly how you want the computer to do something. These statements can involve decisions, operations, input/output, and so on.

Writing the calculator program involves learning how to define variables for use in the calculations, assigning values to the proper variables, deciding what arithmetic operation is needed, performing the calculation, and printing information about the results. For simplicity, you can assume that the program will be executed once for each calculation.

**Inside Information** C provides many ways for you to execute program statements more than once. There are a variety of loop structures (for, do, while) as well as macro facilities and other ways of instructing the program to continue function execution until an end point is reached.

### Example Program: Tasks and Code

The flow of tasks for the calculator program can be summarized as follows:

- asking the user for two numbers
- asking the user for the arithmetic operator
- deciding which calculation to perform
- performing the operation
- displaying the results.

The C functions needed to perform these tasks are

- get a number from the keyboard
- convert the number from a character or string of characters to an integer so the computer can use it in arithmetic operations
- get an operator character from the keyboard
- print the results.

The functions involved can be supplied as follows:

"get a number"
: You will write this one.

"convert character to integer"
: Supplied by C library.

"get a character"
: Supplied by C library.

"print results"
: Supplied by C library.

From this list, you know that three of the functions are already written; you only need to include a reference to the correct library and then call the function when you need to perform that task in the program.

Other C code needed includes a way to declare variables, assign values, make decisions, and perform the calculations.

Here is the complete C code needed to accomplish the program tasks. Comments appear between the /* and */ delimiters. Read through the statements involved and see if you can follow how each program task is performed by identifying the variables and statements used for decisions, calculation, and input/output operations. After looking at the program as a whole, we'll build each section and discuss how each part of the program code contributes to the overall application.

```
/*  Calculator program : add, subtract, multiply, and divide two  */
/*  integer numbers.  Print operands, operator, and results.      */

/*  In the preprocessing step two standard C libraries   */
/*  are included in the program.                         */

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

/*  The main function begins here.  */
main()
{

/*  Declare variables to hold the values for the        */
/*  operands and operator involved in the calculation.  */

int num1, num2, oper;

/*  Print a welcome message and ask the user to enter   */
/*  the first number.                                   */

printf( "Welcome to the Handy-Home Calculator  \n " );

printf( "Please use with integers only.    \n \n " );
printf( "Enter the first number :  " );


/*  The following assignment statement takes the value returned  */
/*  by the getnum function and assigns it to 'num1'.  The same   */
/*  code is used to acquire the second number.                   */

num1 = getnum();

printf( "  \n Enter the second number : " );
Num2 = getnum();

/*  The user is asked for the operator.  The C library function  */
/*  getchar() is used to retrieve the operator. The operator is  */
/*  then assigned to the 'oper' variable.  'oper' is defined as  */
/*  an integer earlier in the program because the ASCII numeric  */
```

```
/*  code for the characters (+, -, *, x, /) is returned.          */

printf("\n Enter the operation to be executed + - * or x, / : ");
oper = getchar();

printf( " \n "); /* New line. */

/*  The following statements do the actual calculations.  The     */
/*  switch structure lets the program branch to perform the       */
/*  correct calculation based on the value of 'oper'.  Once the   */
/*  operator is determined, the calculation is performed in the   */
/*  context of the print function, printf(). The %d indicates     */
/*  to the print function that a decimal number is going to be    */
/*  substituted and printed.  The actual numbers involved         */
/*  follow the %d format.                                         */

switch (oper) {
   case '+' : printf("%d + %d = %d\n ",num1,num2,num1 + num2);
              break;
   case '-' : printf("%d - %d = %d\n ",num1,num2,num1 - num2);
              break;
   case '*' :
   case 'x' : printf("%d * %d = %d\n ",num1,num2,num1 * num2);
              break;
   case '/' : if (num2 == 0) printf("I can't divide by zero.\n");
              else printf("%d / %d = %d\n",num1,num2,num1 /
              num2);
              break;
   default  : printf("I don't understand what %c means.\n",oper);
              }

/*  Main function ends with the following closing brace.  */
}


/*  getnum() is a function you write. It is used by main() to  */
/*  get a number from the keyboard.                            */

getnum()
{
char buff  [ 80 ];
gets( buff );
return( atoi( buff ) );
}
```

If you were to enter the numbers 10 and 2 and the multiplication operator (*) at the program prompts, you would see the following output:

```
Welcome to the Handy-Home Calculator
 Please use with integers only.

 Enter the first number :  10
 Enter the second number : 2

 Enter the operation to be executed + - * or x, / :  *

10 * 2 = 20
```

### A Closer Look

Now that you have an overall idea of the program, we can go back and construct the code from the beginning. (You can use the example as provided on the diskette for the Student Compiler, but it's better practice to key in the statements yourself.) Places where you can use the Student Compiler editor's template feature are noted as you build the program.

The first function coded is `main()`. Before the line calling `main()`, use the C `#include` statement to code the names of two libraries needed for the prewritten C functions you are using. The `#include` statement makes these libraries available to your program. (Function libraries are covered in detail in the *Student Compiler* Reference Guide.) Other initial "housekeeping" code includes declaring (announcing their presence to the compiler) the variables needed for the calculations and printing a welcome message on the screen. (Using the editor for the Student Compiler, you can begin this section by selecting the template for the `main()` function.) Your code to this point looks like this:

```
/*  This is a C comment line - announcing the Calculator program  */

#include <stdio.h>      /* standard I/O library.      */
#include <stdlib.h>     /* standard utility library.  */

main()                  /* main function call.                    */
{                       /* all functions begin with a right brace.  */

   int num1, num2, oper;   /* declare the variables you need.       */

                           /* print a welcome message.              */
   printf( "Welcome to the Handy-Home Calculator \n " );
   printf( "Please use with integers only... \n \n " );

}                       /* all functions end with a left brace.     */
```

Note that the standard C print function, `printf()`, takes care of all the output file-handling details for you; you tell the compiler what you want to print and, optionally, how you want it printed (the \n is called the *newline* character in C).

Next, you can add the code to print messages on the screen that request the numbers and operator involved in the calculation. At this point in the program, two new functions are needed. The function `getnum()` is the one you will write; `getchar()` is a standard C function for getting a character from the keyboard. (As we're building the program, only new items are commented, but all comments should be included in the program's final form.)

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

main()
{

int num1, num2, oper;
printf( "Welcome to the Handy-Home Calculator \n " );

printf( "Please use with integers only...\n \n " );
printf( "Enter the first number :  " );

num1 = getnum(); /*  The getnum() function returns a number that      */
                 /*  is assigned to 'num1' and 'num2', the operands.  */
printf( " \n Enter the second number : " );
num2 = getnum();

printf( " \n Enter the operation to be executed + - * or x, / :"  );
oper = getchar();    /*  getchar() returns a character operator.  */

printf( " \n "); /* Skip a line. */

}
```

Next, you can add the decision structure. This is done by using the C `switch` statement. This statement allows you to choose the appropriate calculations based on the value of the variable `oper` (the available operands are +, -, * or x, and /) entered by the user. Using the Student Compiler's template feature, you can build the code by selecting the `switch` and `if-else` options.

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

main()
{

int num1, num2, oper;
printf( "Welcome to the Handy-Home Calculator \n " );

printf( "Please use with integers only...\n \n " );
printf( "Enter the first number :  " );
num1 = getnum();
printf( " \n Enter the second number : " );
num2 = getnum();

printf("\n Enter the operation to be executed + - * or x, / :");
oper = getchar();

printf( " \n ");
```

```
/*  A switch in C allows the program to test the current value of  */
/*  a variable and then branch to perform some task based on that  */
/*  particular value.                                              */

switch (oper)
   {
   case '+' : printf("%d + %d = %d\n ",num1,num2,num1 + num2);
              break;
   case '-' : printf("%d - %d = %d\n ",num1,num2,num1 - num2);
              break;
   case '*' :
   case 'x' : printf("%d * %d = %d\n ",num1,num2,num1 * num2);
              break;
   case '/' : if (num2 == 0) printf("I can't divide by zero.\n");
              else printf("%d / %d = %d\n ", num1,num2,num1 /
              num 2);
              break;
   default  : printf("I don't understand what %c means.\n",oper);
   }
}
```

Note that in C, you can often combine actions in one statement. For example, the result that you want to print is calculated within the body of the `printf()` statement.

The only thing left to do is to code `getnum()`, the function that retrieves a number from the keyboard and converts it from character form to an integer. (You can use the function template to provide the outline.) This function is added after the closing brace signaling the end of the `main()` function.

```
getnum()
{                          /* function begins */
char buff [ 80 ] ;
                           /*  This is an array to hold the number.   */
                           /*  In C, more than one character is seen  */
                           /*  as a string.  So the number 10 is      */
                           /*  a string of two characters, 1 and 0.   */
                           /*  This array has room for 81 characters  */
                           /*  since C arrays are numbered beginning  */
                           /*  with 0.                                */

gets( buff );              /*  C library function to "get a string"   */
                           /*  and put it into a buffer (the array).  */

return( atoi( buff ) );    /*  C library function to convert the      */
                           /*  string of characters into one integer  */
                           /*  number.                                */

}                          /*  function ends.  */
```

### Trying the Program

As you compile and execute this example, you can use the debugger feature of the Student Edition compiler to follow the program flow. **Figure 2** provides an overview of what you would see as the program executes.

```
main()    entry
          printf()          Welcome to the Handy-Home Calculator
          printf()          Please use integers only...
          printf()          Enter the first number :
          getnum()

getnum()  gets( buff )      (User enters num1 --> 10 )
                            (Buff contains a character string "10")
          return(atoi(buff) (The character string "10" is converted
                             to the integer 10 and returned to the
                             calling function, main() )

main()    printf()          Enter the second number :
          getnum()

getnum()  gets( buff )      (User enters num2 -->  2 )
          return(atoi(buff) ("2" is converted from character to
                             integer and returned to main() )

main()    printf()          Enter the operation to be executed...
          getchar()           (User enters +,-,* or x, / -->  *)
          printf()            (Space one line on the screen)
          if ('oper' == '+')  (  'oper' is not a '+' )
          if ('oper' == '-' ) (  'oper' is not a '-' )
          if ('oper' == '*' ) (  'oper' IS a '*' so the action clause
                                 of the case statement is executed )
          printf()          (The result, 20, is printed)
                            (Program ends.)
```

**Figure 2** Calculator Program - Flow of Statement Execution

### Module Summary

In this module, you learned about the basic structure of a C program and used some basic C statements. The example illustrated that a C program consists of one or more functions and that the C language provides operators and syntax necessary to perform the program tasks involved.

### New C Tools

| Concepts | Code |
|---|---|
| program structure | main() |
| C functions | #include |
| basic statements and expressions | printf() |
| | switch |
| | getchar() |
| | atoi() |

### End of Sample Tutorial Module

This concludes the tutorial example. Remember that the actual tutorial takes many pages to properly explain the concepts and code presented here. The sample module gave you a quick introduction and overview of what C looks like and how you can take advantage of the SAS/C Compiler, Student Edition package to learn the C language.

## CONCLUSION AND SUMMARY

This paper has provided background on the SAS/C Compiler, Student Edition. A general overview of what a compiler is and does was given as well as introductory background on the C language. The tutorial portion of the paper illustrated how the Student Edition compiler can help you learn to program using C.

In summary, the SAS/C Student Edition compiler and tutorial provide an integrated, task-oriented approach to learning C. This is accomplished both by running tutorial examples and by using compiler features. Compiler features such as an editor with a pull-down menu and function keys, templates for common C language structures, an extensive library of C functions, and a source-level dedugger simplify the process of writing C source code. Since all of this is presented in a very fast "load and go" execution environment, you can read about a C language topic and at the same time step through the execution of the sample program. All these compiler features help you use the tutorial and compiler to concentrate on learning C rather than the mechanics of writing and executing programs.

### End Notes

* The tutorial can be used with any C compiler or computer system. The tutorial notes areas where there may be differences between compilers, operating systems, or the ANSI Standard for the C language.

** The example module is a composite of the first several tutorial modules. In the actual tutorial, important C topics such as data types, operators, precedence, function parameters, and return values are all covered in separate modules in much greater detail.

## REFERENCES


American National Standards Committee (1988), *Proposed American National Standard for Information Systems-Programming Language C*, Document Number X3J11/88-090, Washington, D.C.: X3 Secretariat: Computer and Business Equipment Manufacturers Association.

Harbison, Samuel P. and Steele, Guy L., Jr. (1987), *C: A Reference Manual*, 2nd Edition, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Kernighan, Brian W. and Ritchie, Dennis M. (1978), *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Martin, James (1985), *Fourth-Generation Languages, Volume I Principles*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Price, Jonathan (1984), *How to Write a Computer Manual*, Menlo Park, CA: Benjamin/Cummings Publishing Company.

SAS Institute Inc. (1988), *SAS/C Compiler, Student Edition*, Cary, NC: SAS Institute Inc.

Seidman, Arthur H. and Flores, Ivan (eds.) (1984), *The Handbook of Computers and Computing*, New York: Van Nostrand Reinhold Company.