C'99 #4.0
User's Manual

Final TI-99/4a Edition #4.0: 01.01.1988

Software and Documentation written and adapted by
Clint Pulley

Contents

User-supported Software ("FAIRWARE")

The C'99 compiler, libraries, associated software and documentation are provided for your use and that of your friends and/or colleagues.

You are encouraged to distribute C'99 freely provided, you charge no more than media and reasonable distribution costs. All copies of the release diskettes must include this disclaimer.

If you are using C'99 and find it of value, your donation ($20.00 suggested) to the author will be appreciated and will help him to support further development of this product. Contributing users will be placed on a mailing list and advised of new releases.

If you develop useful applications using C'99 you may market them provided that the software and documentation acknowledge the use of C'99. The author would appreciate receiving a complimentary copy of any such programs.

Please address all correspondence to:

Clint Pulley

38 Townsend Avenue

Burlington, Ontario

Canada L7T 1Y6

(416) 639-0583 (home)

Source TI7395

CompuServe 73247,3245

GEnie C.PULLEY

Requests for copies of the C'99 release diskettes should include two formatted diskettes (SS/SD please) in a mailer and $1.00 for return postage. If you already have a version of C'99, please indicate the version number in your letter.

This software carries no warranty, either written or implied, regarding its performance or suitability. Neither the author nor any subsequent distributor accepts any responsibility for losses which might derive from its use.

Chapter 1

INTRODUCTION

C'99 is based on small-C version #1 which was published by Ron Cain in Dr.Dobb's Journal No.45, May 1980. Many enhancements were adapted from Jim Hendrix's small-C #2. Small-C is a useful subset of the C programming language. The compiler produces assembler source code as its output. This code is assembled to produce an object file which is loaded together with all required libraries and run.

C'99 was designed to run in the Editor/Assembler (cartridge) environment on the TI-99/4A computer. Since a number of Extended Basic loadable Editor/Assembler "simulators" are now available, this version of C'99 has been modified to be useable with these products. The compiler and generated programs have been run successfully with FUNLWRITER V 3.0 and BEAXS.

C'99 has these features:

- It supports a subset of the C language.
- Most of the compiler is coded in C'99.
- It is syntactically identical to standard C.
- It produces assembler source code rather than an object file.
- It is a stand-alone single pass compiler.
  It does its own syntax checking and parsing.
- It can compile itself.

Although C'99 lacks many of the features of standard C systems and produces code which is less than optimal, it is, in the opinion of the author, a worthwhile addition to the software repertoire of the TI-99/4A computer. For the first time a structured language with a true compiler is available to TI users. C'99 is sufficiently powerful for the development of utilities, text processors, database systems and games. Since introduction of C'99 over two years ago, many useful programs have been written by users of this language.

Extensive testing of version #4.0 by the author and several advanced C'99 users has not revealed any glaring errors. If you find bugs or wish to suggest improvements, please drop the author a line or leave a message on The Source (TI7395), Compuserve (73247,3245), or GEnie (C.Pulley).

C'99 was developed on a 1983 (black) TI-99/4A with 32K memory expansion, TI (later Myarc) disk controller, two SS/SD drives, RS232 interface and a Roland printer. The final changes for version #4 were developed on a Myarc 9640 computer in 99/4A emulation mode. The Editor/Assembler software environment was used for all software development. All code was written or adapted by Clint Pulley.

This manual assumes a knowledge of standard C or the availability of a suitable reference. The file C99SPECS, found on the release diskette, identifies the features of C which are available in C'99.

Chapter 2

USING C'99

2.1 Entering the source program

Input to the compiler must be a DISplay/VARiable-80 disk file or (for test purposes) the console keyboard. The standard TI editor is normally used for this purpose. If TI-Writer is preferred, be sure to save the source program with the Print File option to avoid getting a line of TAB-data at the end of the file.

2.2 Compiling the source program

From the Editor/Assembler menu, select the Load Program File option. Just press Enter or type in DSKx.UTIL1, where x is the diskette drive containing the compiler disk. When the compiler has been loaded it will identify itself and ask about a number of options. The questions asked are:

```
Include c-text?  [n]
```

This provides the option of including the C'99 source code as comments in the output file. If your response is y or Y each line of C source will appear in the output file preceded by an "*" asterisk. This option should not be selected for large programs as it will result in a very large output file. The default response (n) will result from pressing the enter key.

```
Inline push code?  [n]
```

The C'99 compiler normally generates a subroutine branch to push a value onto its stack. This results in fewer instructions being generated, but execution speed suffers slightly. If maximum performance is required, reply with y or Y. This will cause the compiler to generate the two instruction push sequence inline.

The compiler will then prompt for the input and output filenames. In each case, respond with the full filename (in upper or lower case). If C'99 is unable to open a file it will display "Bad filename try again" and prompt for another name. If the response is a null-name (pressing enter only) that file will map to the keyboard or screen. This may be useful for providing a quick check of a short program. Screen output will pause when a key is pressed.

The compiler will now proceed to process the source program. As each function header is encountered, the first six characters of the function name are displayed on the screen. If it is desired to abort execution at any time, press FCTN-4 (CLEAR). This will terminate the compiler and close all files.

If the compiler encounters an error in your program, it will display an error message on the screen and pause. After noting the error, press enter to resume compilation.

When the compiler has finished, it will display the number of source lines processed, symbol table usage, and the number of errors encountered. It will then ask if more compilations are to be done. A response of y or Y will restart the compiler, a response of n or N will exit. If the Editor/ Assembler cartridge is being used, exit will be to the Ed/Asm screen. Otherwise, exit is to the power-up screen.

2.3 Assembling the compiler output

The output file should be assembled using the TI Assembler (option 2 from the Ed/Asm menu). The R option is not required since C'99 generates numeric register references. This reduces the size of the output file and speeds the assembly slightly.

2.4 Loading the program and libraries

The object file from the assembly may be loaded using the Load and Run option from the Ed/Asm menu. After loading the object file the CSUP library must always be loaded. If the program uses file I/O and contains the statement:

```
#include dsk1.stdio
```

then the CFIO library must also be loaded.

If other external functions or libraries have been referenced by your program they must also be loaded at this time.

2.5 Running the loaded program

When the last file has been loaded, press enter to display the program name prompt. All C'99 programs begin execution at the entry point START. When this has been entered, your program will (hopefully) execute correctly. At program exit the message "C99 Exit/Rerun? (N/Y)" may be displayed. This ensures that the screen will not be cleared at the instant. The program stops and provides the option to rerun the program. Press y or Y to rerun the program, n or N to exit to Ed/Asm.

Chapter 3

ERROR MESSAGES

When C'99 detects an error in the source program it displays the error on screen and pauses. The error display is of the form:

`ERROR` : *description*

displayed line of source code

with ^_ *(pointer to approximate location of error)*

In most cases the description, although brief, is self-explanatory. Among the more cryptic are:

- `NOT AN LVAL` : In C jargon, an lvalue is an expression which may be assigned a value. If a and b are variables, a is an lvalue, a+b is not.
- `TABLE OVERFLOW` : The C'99 compiler contains a number of tables. The capacity of these tables is specified in the C99SPECS file.
- `OUT OF CONTEXT` : Some keywords such as break and case can only be used within loops or switches.
- `LINE TOO LONG` : Every line in your program may not exceed 80 characters, even after macro substitution.

After noting the error, press enter to resume compilation.

If 40 errors are encountered, compilation is terminated.

__Notes:__

The source line displayed has been pre-processed, so all multiple spaces have been removed, all names have been truncated to six characters, and all macro substitutions have been performed.

The error handling in C'99 has been designed to minimise the number of spurious error messages generated. This has resulted in one shortcoming - if a statement contains more than one real error, only the first is reported.

Chapter 4

LIBRARIES AND INCLUDE FILES

The object libraries provided with this release are:

CSUP - The compiler support library. It contains the initialization, exit, and direct support (C$) functions required by all C'99 programs as well as console I/O functions.

CFIO - The file input/output library. It contains the file tables and all file I/O functions.

The include files provided with this release are:

CONIO - I/O definitions for console functions only.

STDIO - I/O definitions for console and file functions as well as extern specifiers for all functions in CFIO. If STDIO is included in a program, CONIO must not be included or duplicate definitions will result.

Chapter 5

LIBRARY FUNCTIONS

Each function is introduced by a sample call. If a function returns a value, an assignment is shown. You may, of course, discard the function result. Arguments must be of the same type as the sample.

The following declarations specify the type of all variables and arguments used in the sample calls.

```
int  b, c, f, row, col, key, unit, len, recno;

char buff[81];

char *filename, *mode, *name, *string;
```

5.1 Functions in CSUP

5.1.1 Read one character from the keyboard.

`c=getchar();`

Waits for a key to be pressed and returns the character value. The character is echoed to the screen. If the character is "CR" (Enter), the screen spaces to the start of a new line and a value of "EOL"= 10 is returned. If the character is CTRL-Z, "EOF"= -1 is returned.

5.1.2 Write one character to the screen.

`c=putchar(c);`

Writes the character whose ascii value is c to the screen. If c==10 (EOL), the screen spaces to the start of a new line. If c==8 (BS), the cursor is backspaced.

If c==12 (FF), the screen is cleared and the cursor is homed.

If c represents a non-printable character, a "\" is echoed. The value of c is returned.

5.1.3 Read a line from the keyboard.

`c=gets(buff);`

Reads one line from the keyboard into a character array. The line is terminated with ENTER, CTRL-Z or the 80th character. The array is assumed to be 81 characters characters long and a NULL-byte (0) is appended to the end of the string. A value of `buff` is returned unless CTRL-Z is pressed. In that case, 0 (NULL) is returned. Use of the backspace key (FCTN-S) for inline editing is supported.

5.1.4 Write a string to the screen.

`puts(string);`

Writes a string to the screen, stopping when it finds a NULL-byte. The NULL-character is not written. The cursor is not spaced to the start of a new line unless newline (\n) is encountered.

5.1.5 Locate the cursor on the screen.

`locate(row,col);`

Places the cursor at the screen location specified by row and column. Subsequent console I/O will start at the new cursor location. Row and column numbering start at 1 as in TI Basic. The validity of row and col is not checked.

5.1.6 Exit the program.

`exit(c);` or `abort(c);`

Branches to the C'99 exit function which closes any open files. The value of c may be between 0 and 7. If c==0, the normal exit message is displayed. If c==7, the program terminates immediately. Otherwise the value of c is displayed in an error message. The exit(0) function is also executed when function main terminates.

5.1.7 Check keyboard status.

`key=poll(c);`

Scans the keyboard and returns the key value (if one is pressed) or 0. If c!=0, the program will pause while a key is down. If FCTN-4 (CLEAR) is pressed, the program will branch to the C'99 exit function.

5.1.8 Change screen color.

`tscrn(f,b);`

Changes the text mode screen colors to f (foreground) and b (background). The Basic color number convention is used.

5.2 Functions in CFIO:

Note: In most of the file I/O functions which reference the argument "unit", the operation will default to the corresponding console I/O function if the value of unit is <= 0. For this reason, the values for stdin, stdout, and stderr as defined in STDIO are -1, -2 and -3.

5.2.1 Open a file.

`unit=fopen(name,mode);`

Opens the named file in the specified mode. Both name and mode must be strings or pointers to strings. Currently supported modes are:

display/variable	display/fixed	display/relative
"r" - read	"R" - read	"I" - read
"w" - write	"W" - write	"O" - read/write
"u" - update	"U" - update	
"a" - append		

In the mode parameter string, the mode character may be followed by a 1-3 digit record length. If this is omitted, a default length of 80 is assigned. If the file is opened for input and a record length of zero is specified, the actual record length of the existing file is utilised. The function `ferrc(unit)` can be used to obtain the record length.

A unit number is returned for use with the file I/O functions. This unit number must not be altered. If the open fails, NULL (0) is returned. No more than four files may be open simultaneously and no more than three may be disk files. Filenames may be upper or lower case and must not exceed 26 characters in length.

5.2.2 Close a file.

```
c=fclose(unit);
```

Performs the appropriate file closing action and makes the unit available for another file. In the case of output files being written with putc, an incomplete line is lost! This function returns NULL if the close fails and non-null if it succeeds.

All open files are closed automatically if a program terminates normally.

5.2.3 Delete a file.

```
fdelete(filename);
```

Deletes the file specified by filename, which must be a string or pointer. No error conditions are returned.

5.2.4 Read one character from a file.

```
c=getc(unit);
```

Reads and returns the next character from the file corresponding to unit.

If the end-of-line is reached a value of 10 (EOL) is returned.

If the end-of-file is reached, a value of -1 (EOF) is returned. If an error occurs, -2 (ERR) is returned.

5.2.5 Write one character to a file.

`c=putc(c,unit);`

Writes the character whose ascii value is c to the file.

If c==10 (EOL), the actual write operation occurs. The value of c is returned. If an error occurs, -2 (ERR) is returned.

5.2.6 Read a string from a file.

`c=fgets(buff,col,unit);`

Reads one line from the file into a character array. At most, col-1 characters will be transferred. A NULL byte is appended to the end of the line. If a partial line is transferred, the remainder of the line is discarded. If unit<= 0, `gets` is called and the value of col is ignored. This could result in buffer overflow. A value of buff is returned. If an end-of-file or error condition occurs, NULL is returned.

5.2.7 Write a string to a file.

`c=fputs(string,unit);`

Writes a string to a file, stopping when it finds a NULL byte character. Imbedded EOL characters act as record terminators, so multiple records can be generated by a single call to fputs. A value of buff is returned. On end-of-file or error, NULL is returned.

5.2.8 Test for end-of file.

`c=feof(unit);`

Returns a TRUE value if the next read from unit would return an end-of-file error condition. Returns FALSE otherwise.

5.2.9 Read a record from a file.

`c=fread(buff,len,unit);`

Reads the next record from the file into the buffer area starting at buff. At most, len bytes will be transferred. A NULL byte is NOT appended. If a partial record is transferred, the remainder is discarded. The actual number of bytes transferred is returned. If an end-of-file or error condition occurs, -2 (ERR) is returned.

5.2.10 Write a record to a file.

```
c=fwrite(buff,len,unit);
```

Writes a record of len bytes from the buffer area starting at buff. If len is greater than the maximum record length for the file the record is truncated. No special action occurs for NULL or EOL bytes. The actual number of bytes transferred is returned. If an error condition occurs, -2 (ERR) is returned.

Note: fread/fwrite does NOT default to the console !!

fread/fwrite provide a binary I/O capability for applications such as printer dot-graphics since all bytes are transferred regardless of value. If the buffer area is comprised of some consecutive global variables and arrays then transfers can be made directly from/to the variables and arrays. If the first element of the buffer area is a scalar variable then its address (&var) must be used in the function call.

5.2.11 Set record number.

```
fseek(unit,recno);
```

Sets the record number for the next I/O operation (fread or fwrite) on unit. This function provides a random access capability for files opened as relative. If a single fseek is followed by multiple fread or fwrite operations access becomes sequential starting with recno.

5.2.12 Get error code.

```
c=ferrc(unit);
```

If the previous I/O operation resulted in an error, returns the error code. This function should be used only when an error has occurred. The returned value is meaningless otherwise. Error codes are listed in the TI-99/4A reference manual. This function cannot be used after fopen errors since unit is not valid at that time. If used immediately after a successful fopen, ferrc returns the actual record length of the file.

5.2.13 Rewind a file.

```
rewind(unit);
```

If a disk file is open for read or append it is rewound. All other cases are ignored.

Chapter 6

ASSEMBLY LANGUAGE INTERFACE

Interfacing to assembly language is relatively straightforward. The "#asm ... #endasm" form allows the placing of assembly source code directly into the program. Since the compiler considers it to be a single statement, it may be used as:

```
while(1) #asm ...  #endasm
```

or

```
if(expression) #asm ...  #endasm else ...
```

In actual program coding, the #asm directive must be the last item on a line and the #endasm directive must appear on a line by itself. Since the compiler is free-format otherwise, the expected format is:

```
if(expression)
  #asm
  ....
  ....
  #endasm
else statement;
```

A semicolon is not required after #endasm.

Assembly code within the "#asm ... #endasm" form has access to all global symbols and functions by name. It is the programmer's responsibility to know the data type of the symbol (whether "int" or "char" implies word or byte access). Stack locals and arguments may be retrieved by offset.

The push-down stack used by C'99 is located in the upper part of the low (8K) bank of the TI-99/4A memory expansion. Register R14 in the C'99 workspace is reserved as the stack pointer. The stack begins at >3FFE and grows towards >2678. External assembly language routines accessed by function calls from C code may use registers R0 thru R7. They may push items on the stack, but must pop them off before exit. It is the responsibility of the calling program to remove arguments from the stack after a function call. Since arguments are passed by value, the arguments on the stack may be modified by the called program.

Chapter 7

MEMORY UTILIZATION

7.1 PAD Usage.

C'99 reserves PAD locations >8300 - >832F for its workspace and support code. Other PAD locations not used by console routines are available to the programmer. In particular, locations >8330 - >8348 can be used to store frequently accessed global variables by use of AORG or DORG directives.

The C'99 workspace is at >8300. Register utilization is:

R0-R7	temporary storage
R8	primary computation register
R9	local address register
R10	address of the least-significant byte of R8
R11	return address for BL instruction
R12	address of recursive subroutine call routine
R13	address of recursive subroutine return routine
R14	the stack pointer
R15	first word of the PUSH routine (hence BL 15)

7.2 Memory Expansion Usage.

The entire 24K bank of memory is available for program usage. The 8K bank contains the standard Editor/Assembler utilities (>2000 - >2676). As previously mentioned, the C'99 stack grows down from >3FFE. Since typical stack usage is a few hundred bytes, the intervening space is available.

Since no indication of stack/program overlap is provided in this version of C'99, very large programs could crash. However, the C'99 compiler which uses all of the 24K bank and much of the 8K bank is able to compile itself successfully, so this problem may never arise for most users.

7.3 VDP Ram Usage.

Aside from the areas normally used in the Ed/Asm environment, C'99 reserves VDP memory locations >1B70 (>1B5D for the delete function) thru >1FFF for file I/O requirements. This area was chosen to permit implementation of bit-map graphics.

Chapter 8

STACK USAGE

C'99 makes extensive use of the previously-mentioned push-down stack for temporary storage. Function arguments are pushed onto the stack as they are encountered between parentheses, so the last argument is at the "top" of the stack. This inverse order is somewhat unconventional. After all arguments have been pushed on the stack, the return address is pushed on by the recursive call code accessed via R12. Since the stack grows downwards in memory, the last argument value is located two bytes above the stack pointer's current contents at function entry.

As specified in the C language definition, parameter passing is "call by value". If X and Y are global variables, the compiler generates the following code for this C statement:

```
X=function1(X,Y,zf());

MOV @X,8      value of X to primary register
BL 15         push onto stack
MOV @Y,8      ditto for Y
BL 15
BL *12        recursive call zf (subroutine)
DATA ZF       every function value is returned in R8
BL 15         push value onto stack
BL *12        call function1
DATA FUNCTI   note 6 characters used
AI 14,6       restore stack pointer
MOV 8,@X      primary reg to X
```

As function1 is entered, the stack contents are:

```
        .
        .
        .
 -----------
 value of X
 -----------
 value of Y
 -----------
 zf fcn value
 -----------
 return addr     <=== stack pointer (R14)
 -----------
```

In this case, the value of Y could be accessed by "`MOV @4(14),8`".

Local variables allocate as much stack space as needed and are assigned the current value of the stack pointer (after allocation) as their address. The compiler ensures that each variable is located on a word boundary.

The declarations:

```
    int z;
    char array[5];
```

generate:

```
    AI   14,-8
```

which allocates space on the stack for 8 bytes (not initialized). References to z will be made to stack pointer+6. Note that the stack pointer changes by 8 (not 7) bytes, ensuring that the following instruction falls on a word boundary.

Until the stack pointer is altered again, , array[0] is at *14, array[1] is at @1(14), array[2] is at @2(14), etc. For this reason, imbedded assembly language code using "`#asm ... #endasm`" cannot access local variables by name, but must know their location relative to the stack pointer's current contents.

Chapter 9

PROGRAMMING INFORMATION.

- When a C'99 program begins execution, the screen is in text mode (40 characters/line) displaying white characters on a dark blue background. The function `tscrn` provides a means of changing the default color. It is possible to access VDP registers and memory from C'99 by using the appropriate values with pointers.

- The logical operators "`&&`" and "`||`" (with left-to-right evaluation and early dropout) are available in C'99.

  The bitwise operators "`&`" and "`|`" will usually yield the correct results in logical expressions, but forms such as "`if(i&j)`" should be avoided as erroneous results may be produced!! (eg. if `i=1`, `j=2` then `i&j=0`).

  C'99 follows the usual convention in using a non-zero value to represent a true condition and a zero value to represent a false condition.

- Global variables result in more efficient code than local variables but they cannot be used in recursive situations.

  In addition, global variables (especially arrays) increase the size of a program module.

- Functions may be passed the names of other functions as arguments for indirect calling.

  The dummy argument for a function name must appear in the argument list as a simple name and must be declared as an integer pointer. Calls should be coded as "`fcn()`" and not as "`(*fcn)()`".

- Since C'99 programs are self-contained, they may be saved as program files. Two object files, C99PFI and C99PFF, have been placed on the release diskette to facilitate program file creation. C99PFI must be loaded before user programs and libraries since it defines the entry points SFIRST and SLOAD and contains code to check for the presence of the Editor/Assembler cartridge and load the standard utilities from the Ed/Asm GROM. C99PFF must be loaded after all programs and libraries as it defines the entry point SLAST.

- If program files are run using an Editor/Assembler simulator, they will only run if the standard utilities are available. The two simulators tested both satisfied this requirement. Since the possibility of over-writing the simulator code exists, exit is to the power-up title screen.

- If the rerun option is chosen at program termination, global variables are NOT reset to their initial values. Since this may cause undesirable results in some programs, the ability to bypass the rerun option is provided by exit(7).

- C'99 has consistent handling of GROM addresses and GPL subprogram linkage. The GROM address is saved at program start and restored at program exit. C$GPLL, a special GPLLNK function which will function in any run mode, is included in CSUP. This function is used as follows:

```
extern C$GPLL()
  . . .
  . . .
#asm
 BL @C$GPLL
 DATA xxx   <- the address of the required routine, as
#endasm        specified for GPLLNK in the Ed/Asm manual.
```

- A modest degree of incompatibilty exists between this version of C'99 and the versions #2.0/2.1. :

  Indirect function calls are handled differently, so any library functions using this feature must be recompiled. The new CSUP library must be used with all programs compiled by C'99 #4.0, but this library is compatible with functions compiled by earlier versions #2.0/2.1. Since the object code produced by #4.0 is more efficient and compact, consideration should be given to recompiling all frequently used programs and functions.

Chapter 10

C99 V4.0 COMPILER SPECIFICATIONS

10.1 Features of C'99 (Revised 88/01/01)

C'99 currently supports:

1. Data type declarations of:

   "char" (8 bits)

   "int" (16 bits)

   pointers to either of the above by using an "*" before the variable name.

   arrays of pointers to either of the above by using an "*" before the array name.

   initializers on declarations for global variables.

   The storage class of a declaration is implicitly determined by its position. Declarations occurring outside of any function are global (static) while declarations occurring inside a function definition are local (auto).

2. Storage class specifier of:

   "extern" (provides linkage to functions and global variables defined in other object modules by generating REF assembler directives.)

3. Arrays:

   One- or two- dimensional arrays may be of type char or int.

4. Pointers:

   Local and static pointers can contain the address of char or int data elements.

5. Program control:

   if(expression) statement;

   if(expression) statement; else statement;

   while(expression) statement;

   do statement while(expression);

   for(expression1;expression2;expression3)statement;

   switch(expression) statement;

   goto label;

   case constant :

   default :

   break;

   continue;

   label :

   return;

   return expression;

   ; (null statement)

   { statement; statement; ... } (compound statement)

6. Expressions:

   unary operators:

   "-" (minus)

   "*" (indirection)

   "&" (address of)

   "~" (ones complement)

   "!" (logical negation)

   "++" (increment, either prefix or postfix)

   "– –" (decrement, either prefix or postfix)

   binary operators:

   "+" (addition)

   "–" (subtraction)

   "*" (multiplication)

”/” (division)

”%” (modulo, ie. remainder from division)

”|” (bitwise inclusive or)

”^ ” (bitwise exclusive or)

”&” (bitwise and)

”==” (test if equal) ”!=” (test if not equal)

”<” (test if less than)

”<=” (test if less or equal)

”>” (test if greater than)

”>=” (test if greater or equal)

”<<” (arithmetic left shift)

”>>” (arithmetic right shift)

”=” (assignment)

expression , expression

logical operators:

”&&” (logical and with left-to-right evaluation and early dropout)

”||” (logical or with left-to-right evaluation and early dropout)

conditional expression:

expression ? expression : expression ;

(the second expression is evaluated only if the first is true, the third is evaluated only if the first is false)

primaries:

array[expression]

function(arg1,arg2,...,argn)

local variable or pointer

global (static) variable or pointer

constants:

decimal number

hexadecimal number

quoted string (”sample string”)

primed string ('a' or 'ab')

(the \ character constant is supported)

7. Compiler commands:

   #define name string

   (pre-processor will replace name with string)

   #include "filename"

   (take input from filename until end-of-file. cannot be nested.)

   #ifdef name

   (compiles following lines if name is defined by #define)

   #ifndef name

   (compiles following lines if name is undefined)

   #else

   (compiles following lines if previous #if... was false)

   #endif

   (ends conditional compilation block)

   #asm ... #endasm

   (allows all code between "#asm" and "#endasm" to be passed unchanged to the assembler.

   this command is actually a statement and may be used as: "if (expression) #asm ... #endasm else ...")

8. Miscellaneous:

   Expression evaluation maintains the same heirarchy as in standard C.

   Pointer arithmetic recognizes the data type of the destination. ptr++ will increment by two if ptr was declared as "int *ptr".

   Pointer compares are unsigned since addresses are not signed numbers.

   Operations which require more than two words of instructions generate calls to routines in the CSUP library to minimize the size of the program. All such support routines have names beginning with C$.

   The underscore character "_" is translated to "#" so that generated labels will be recognized by the assembler.

   Conditional compilation directives (#if...) may be nested to any level.

   The generated code is re-entrant as required by C.

   Each time a function is referenced, the local variables refer to a fresh area of the stack.

   The entry statement (not in standard C) makes a function or global variable name available to other modules by generating DEF assembler directives. Usage is:

entry name1,name2,...;

Note that parentheses and brackets are not necessary in entry statements.

10.2 Limitations of C'99

C'99 does not support:

1. Structures, unions and arrays of more than two dimensions.
2. Nested initializers for two-dimensional arrays.
3. Constant expressions in initializers, case and as array bounds (constants must be used).
4. Data types other than "char" and "int".
5. Function calls returning other than "int" values (returned pointers are valid since they are the same size!)
6. The unary "sizeof" and casts.
7. The assignment operators `+= -= *= /= %= >>= <<= &= ^= |=`
8. Storage class specifiers: auto, static, register, typedef.
9. The use of arguments within a "#define" command.
10. Pointers to anything but char or int.
11. \ followed by ' in a primed string or " in a quoted string.

C'99 programs may have a maximum of:

1. 1000 characters of macro (#define) definitions.
2. 256 global symbols (variable and function names, including functions external to the program).
3. 40 local symbols within a single function definition.
4. 20 simultaneously active loops (while, for, do).
5. 60 cases within any one switch.
6. 4096 characters of literal strings ("string") within a single function definition or initializer.

Other limitations of C'99:

The implementation of indirect function calls is non-standard !! Such calls may not be nested (ie. an indirect call must not have another indirect function evaluation as a parameter) and usage is not as in standard C.

Function names passed as parameters should be declared as integer pointers and calls to such functions should be coded as normal calls: "fcn()", not "(*fcn)()" .

Since C'99 is a single-pass compiler, undefined names are not detected and are assumed to be function names not yet defined. If this assumption is incorrect, an undefined REFerence error will occur when the compiled program is assembled.

Because a single-pass compiler scans the source code only once, very little object code optimization is possible. For example, the statement x=1+2; results in code to add 1 and 2 at runtime.

Names may be of any length but the compiler only recognizes the first six characters. Names used with #ifdef and #ifndef must not exceed six characters.

REF (external reference) directives are generated for functions in the CSUP library only. REFs to functions in other libraries must be provided explicitly using extern.

The include file STDIO provides extern specifiers for all functions in the CFIO library.

A DEF (external definition) directive is automatically generated only for function main. The entry statement will generate DEFs for function and global variable names, making them available to other object modules.

The file I/O library (CFIO) is limited to processing Display type files. Tom Bentley of Ottawa, Ontario has written an excellent I/O function library which supports Internal files.

Global variables which are not specifically initalized have an indeterminate value at program start.

Initializers for two-dimensional arrays may not contain nested row initializers. Values in the initialization vector will be stored row-wise, starting from the first element of the array. Two dimensional character arrays may be initialized with a sequence of strings, each with length of row size-1 to allow for the NULL-byte terminator.

Chapter 11

REFERENCES.

- The C Programming Language by Brian Kernighan and Dennis Ritchie
  Prentice-Hall 1978
- The C Primer by Les Hancock and Morris Kreiger
  McGraw-Hill 1982 (also C Primer Plus ...)
- Learning to program in C by Thomas Plum
  Plum Hall Inc. 1983
- A number of German C'99 users have found quite useful:
  "Das C-Anwender Handbuch", R.Heigenmoser, im Hofacker-Verlag.

Typesetting in LaTeX by T. Brouwer