DTL

Data Transformation Language

Phillip H. Sherrod

www.dtreg.com

DTL is a full programming language built into the DTREG program. DTL makes it easy to generate new variables, transform and combine input variables and select records to be used in the analysis.

Contents

Introduction

Introduction to the DTL Language

DTL is a complete programming language. Using it, you can manage data being analyzed, generate new variables as a function of input variables and select which records are used in an analysis. Although DTL does not have all of the features of languages such as C and Visual Basic you will find that it is a rich language unto itself and includes features not commonly found in other languages such as fully dynamic string variables and a wide selection of built-in library functions.

The syntax of the DTL language is intentionally similar to that of the C programming language. If you have prior experience programming in C you will be able to begin using DTL almost immediately. The following list summarizes the major differences between C and DTL:

- DTL does not have structures or pointers.
- Arguments to functions are passed by copying. On entry the values of calling arguments are copied to the receiving formal parameters. On exit, the values are copied back to the calling arguments.
- DTL supports three data types: **int** (32-bit integer), **double** (64-bit floating point), and **string** (variable length strings).
- The DTL string data type provides fully dynamic strings whose size is determined at execution time rather than by statement declaration. DTL strings can store binary data including the null character.
- DTL provides a substring operator for extracting or changing a portion of a string. There is also a string concatenation operator ('$').
- DTL supports one and two-dimensional arrays. The syntax for declaring the array size and for subscripting elements has the form "*array*[*sub1*,*sub2*]" rather than the C convention of "*array*[*sub1*][*sub2*]". A "resize" function can be used to change the size of an array during the execution of a program. When arrays are passed to functions the size of the array in the function "conforms" to the size of the passed array. Library functions are provided to determine the actual size of an array.
- DTL is very "liberal" with regard to type conversions. Any type of variable may be assigned to any other. Any type of variable or expression may be used as an argument to a function without regard to the type of the formal parameter. String and numeric values may be used together in expressions.

Using DTL For Data Transformations

DTL is a full-featured programming language. Before getting into the detailed DTL language reference, we will look at some typical uses of DTL with DTREG analyses.

The main() function

Every DTL program must have a main() function that is executed by DTREG for each data record. The main() function must contain a return statement that signals DTREG whether the current record is to be used in the analysis or excluded. If the return statement returns a value of 1, the record is used in the analysis. If the return statement returns a value of 0 (zero), the record is excluded from the analysis.

Here is a simple main program that accepts all records:

```
int main()
{
      return(1);
}
```

Here is an example that accepts records that have a value of "M" for Sex and rejects other records:

```
int main()
{
      if (Sex == "M") {
            return(1);
      } else {
            return(0);
      }
}
```

Here is an example that accepts records that have a value of "M" for Sex variable and a value of 65 or greater for Age:

```
int main()
{
      if (Sex == "M" && Age >= 65) {
            return(1);
      } else {
            return(0);
      }
}
```

Here is a main program that accepts about half of the records and rejects half:

```
int main()
{
      if (random() > 0.5) {
            return(1);
      } else {
            return(0);
      }
}
```

Global Variables

A global variable is a variable defined outside the scope of any function; usually, global variables are defined at the top of the program. Global variables can be accessed by any function in the DTL program. Global variables may have any of the three data types, **int**, **double** or **string**. Global variables you define are called *explicit* global variables. Global variables defined automatically by DTREG are called *implicit* global variables.

Implicit Global Variables

DTREG defines implicit global variables for each variable in the input data file. This includes *all* data variables, even variables not designated as predictor, target or weight variables. The implicit global variables are not visible in the DTL source program, but they can be used by the program.

If a variable is specified as categorical in the DTREG model, the implicit definition has type string. If the variable is specified as continuous, the implicit definition has type double. For example, if a data file contains four continuous variables, Age, BloodPressure, Height,

Weight and one categorical variable Sex, then the implicit definitions (which you will not see) would be:

```
double Age;
double BloodPressure;
double Height;
double Weight;
string Sex;
```

The main() function and any other functions in the DTL program can reference these implicit global variables.

In addition to generating a global variable for each variable in the data file, DTREG also generates several other global variables:

```
int RECORDNUMBER;       /* The number of the current data record */
int DOINGSCORE;         /* 1 if scoring, 0 if analysis is being run */
double MISSINGVALUE;    /* Value used to indicate missing value */
```

Any changes your program makes to the values of implicit global variables are *not* used in the analysis. If you want to transform variables, you must define your own global variables as described below and store values into them.

Explicit Global Variables

You can define your own global variables by putting their definitions outside the scope of any function. It is recommended that they be put at the top of the DTL program before main().

Any global variable you define in a DTL program that does not have the "static" declaration will be available as a variable in the DTREG analysis. This is the way you generate transformed variables. For example, the following program generates a new variable, Size, which is the product of two input data variables, Height and Weight:

```
double Size;
int main()
{
      Size = Height * Weight;
      return(1);
}
```

With this DTL program defined, the Size variable will be available for use in the DTREG analysis. The Height and Weight variables also are available.

Here is an example that creates a variable called Republican that is 1 if the value of PartyAffiliation is “R” and 0 if PartyAffiliation is anything else:

```
double Republican;
int main()
{
      if (PartyAffiliation == “R”) {
            Republican = 1;
      } else {
            Republican = 0;
      }
      return(1);
}
```

Here is an example that creates a LogAge variable that is the natural logarithm of the Age variable:

```
double LogAge;
int main()
{
      LogAge = log(Age);
      return(1);
}
```

Here is an example that creates a variable named ZIP3 that has the first three digits of a zip code whose five-digit code is stored in ZIP5. The substring operator, **[***start***:***length***],** is used to extract the first three characters.

```
string ZIP3;
int main()
{
      ZIP3 = ZIP5[0:3];
      return(1);
}
```

Sometimes missing values for numeric variables are coded with values like "999". DTREG uses a special value called "MissingValue" to indicate missing values. Here is an example DTL program that converts input data values of "999" on an Age variable to the internal missing value. The new variable with the transformed values is called NewAge.

```
int main()
double NewAge;
{
      if (Age == 999) {
            NewAge = MissingValue;
      } else {
            NewAge = Age;
      }
      return(1);
}
```

Static Global Variables

Static global variables are used to store information between calls of the main() function for each data record. They also can be used to hold information that must be accessed by multiple functions. Static global variables may *not* be used as variables in the DTREG analysis. To declare a static global variable, put the word "static" in front of the declaration like this:

```
static int FileNumber;
static int Count;
static double LastAge;
static string LastName;
```

Using the StoreData() function to generate data records

The main() function is called for each record in the input data file, and it returns 1 to keep the record or 0 to reject the record. DTL provides a StoreData() function that you can call to generate additional records. Each time you call StoreData(), the current values of the global variables are used to generate a new data record which is included in the analysis. This allows you to generate multiple records from a single input record.

Consider a data set that is to be analyzed using logistic regression. The data set measures the response of patients to varying dose levels of a drug. There are three variables in the input data file, Dose (the amount of the drug), Positive (the number of patients with positive responses) and Negative (the number of patients that did not respond). Hence the implicit global definitions generated by DTREG for the DTL program are:

```
double Dose;
double Positive;
double Negative;
```

The following DTL program defines a new variable, Response, that has the value 1 if the patient responds positively and 0 if the patient does not respond. The DTL program uses the StoreData() function to generate a separate record for each patient. After calling StoreData() the appropriate number of times, it uses the return(0) statement to reject the original record.

```
double Response;  /* Generated variable with 1 or 0 response */
int main()
{
      int count;
      /* Generate the positive response records */
      Response = 1;
      for (count=0; count<Positive; count++) {
            StoreData();
      }
      /* Generate the negative response records */
      Response = 0;
      For (count=0; count<Negative; count++) {
            StoreData();
      }
      /* Reject the original record */
      return(0);
}
```

The StartRun() and EndRun() Functions

The optional StartRun() and EndRun() functions can be used to perform initialization and cleanup in a DTL program.

If your DTL program contains a StartRun() function, it is called once at the beginning of the run before the first data record is processed. It can perform initialization.

If your DTL program contains an EndRun() function, it is called once after the last data record has been read.

In the following example, the DTL program opens an output file in the StartRun() function, writes information about each data record in the main() function and closes the file in the

EndRun() function. Note the use of a static global variable to store the file handle number between iterations.

```
static int FileHandle;

void StartRun()
{
      FileHandle = fopen(“Data.dat”,”wt”);
      return;
}

int main()
{
      fprintf(FileHandle,”%f %f\n”,x,y);
      return(1);
}

void EndRun()
{
      fclose(FileHandle);
      return;
}
```

DTL Language Reference

A DTL program is divided into units called “functions”. A simple program may consist of a single function; large programs are usually divided into multiple functions. It is a good idea to design your functions so that each one performs a single, identifiable task.

Functions can be divided into two groups: user defined and built-in. User defined functions are the ones that you write. As you develop programs you will find that functions you wrote for one program will be useful for other programs and you will begin to develop a library of functions that do common operations. Built-in functions are a standard part of the DTL environment, and you can use them without having to write them. Some examples of built-in functions are those for reading and writing files, computing square roots, and displaying text.

Regardless of the number of functions that you use to create your program, execution begins at the first statement in a function that you must provide named “**main**”.

The following is an example of a very simple DTL program that simply displays the string “Hello world” and then exits:

```
int main()
{
    printf("Hello world\n");
    return(1);
}
```

This example consists of a single function named “main” that has only two executable statements. When the program is run execution begins at the printf statement. When the return statement is executed in the main program the program exits.

Expressions

Numeric constants

Numeric constants may be written in their natural form (1, 0, 1.5, .0003, etc.) or in exponential form, *n.nnn*E*ppp*, where *n.nnn* is the base value and *ppp* is the power of ten by which the base is multiplied. For example, the number 1.5E4 is equivalent to 15000.

If a number contains a decimal point it is considered to be a real (double) value. Real values are stored as 64-bit double precision numbers and have a possible range from 1.7E-308 to 1.7E+308.

A number without a decimal point is an integer constant. Integer constants are stored as 32-bit long values and have the range –2,147,483,648 to +2,147,483,647.

Hexadecimal constants may be written in the form "0X*nnnn*" where "0X" is a prefix indicating that this is a hexadecimal constant and "*nnnn*" are the hexadecimal digits from the set 0..9, A..F, and a..f. For example, the hex constant "0X1A" is equal to the decimal value 26.

String constants

String constants are written by enclosing the string in quote marks. For example, "Hello World" is an example of a string constant. In addition to letters, digits, and punctuation signs, string constants may contain "escape sequences" that represent certain control characters. An escape sequence in a string constant begins with the backslash character, '\', which is followed by one or more characters. For example, the string "abc\n" contains the characters "abc" and a line-feed character which causes the cursor to move to the beginning of the next line when you print the string using the printf function.

The DTL compiler converts these escape sequences to characters in the string as follows:

\a — Causes a bell character (hex 07) to be placed in the string.

\b — Causes a backspace character (hex 08) to be placed in the string.

\e — Causes an escape character (hex 1B) to be placed in the string.

\f — Causes a form-feed character (hex 0C) to be placed in the string. Form-feed characters cause the screen to be cleared. If sent to a hardcopy printer, a form-feed character causes a page to be ejected.

\n — Converted to a line-feed character (hex 0A). When a string with a line-feed character is written to the terminal or a text file, the line-feed character is converted to two characters: carriage-return followed by line-feed. Thus '\n' is the escape sequence that you should place in print string where you want a new line to begin.

\r — Causes a carriage-return character (hex 0D) to be placed in the string. Note: since '\n' generates a carriage-return, line-feed pair, you do not normally need to use the \r escape sequence.

\t — Causes a horizontal tab character (hex 09) to be placed in the string.

\v — Causes a vertical tab (hex 0B) to be placed in the string.

\x*dd* — Causes the two or three hexadecimal digits that follow \x to be converted to a single character. For example, the sequence '\x41' would generate a single character whose ASCII code is hex 41, this is the letter 'A'.

**** — Causes a single '\' character to be stored in the string.

\" — Causes a quote mark character to be placed in the string rather than terminating the string.

For example the statement

```
s = "Phil\n"
```

assigns to the variable ‘s’ a string consisting of the characters “Phil” followed by a new line character.

Note that you must specify “\\” in a quoted string to represent a single ‘\’ character. This is important when writing string that contain file specifications with directories. For example, to specify a string for “C:\WORK\TEST.DAT” you would type “C:\\WORK\\TEST.DAT”.

Variable names

A variable name must begin with a letter or the underscore character (‘_’). Characters in the name after the first may be letters, digits, or the underscore character. A variable name may be up to 31 characters long. DTL variable names are *not* case sensitive. In other words, the variables ‘TIME’ and ‘time’ are the same.

Variables defined outside any functions are called “global variables”. They may be used by any function. Variables declared within a function are “local” to that function and may not be referenced by another function. In fact, multiple functions may use the same names for local variables without conflict since they are separate variables.

Statement labels

A statement label has the same form as a variable name: it may be up to 31 characters long, must begin with a letter or underscore, and it may be formed of letters, digits, and underscores after the first character. If a statement label is specified it must be at the beginning of a statement and must be followed by a colon character. The following is an example of a program that uses statement labels which are ‘top’ and ‘end’ (note, this program could be written more cleanly using a ‘for’ statement):

```
void main()
{
    int i;
    i = 1;
top:    if (i > 5) goto end;
    print(i);
    i++;
    goto top;
end:    return;
}
```

Operators

Arithmetic Operators

The following arithmetic operators may be used in expressions:

```
++       add 1 to a variable
--       subtract 1 from a variable
+        addition
-        subtraction or unary minus
*        multiplication
/        division
%        modulo
** or ^  exponentiation
```

The "++" and "--" operators may be used either immediately before or after a variable name. If they are used before the name, the increment or decrement is performed before the value of the variable is used in the expression. If they are used after the name, the value of the variable before being modified is used in the expression and then the increment or decrement takes place. For example, the sequence:

```
a = 3;
b = 3;
x = ++a;
y = b++;
```

assigns the value 4 to x and 3 to y. At the end of the sequence, both a and b have the value 4.

String Operators

There are three string operators: concatenate, append, and substring.

The dollar sign ('**$**') is the **concatenation operator**. It causes the strings on either side of it to be concatenated together. For example, the statement

```
s = "Hello" $ " " $ "world";
```

causes the three strings, "Hello", " " (a single space), and "world" to be concatenated, producing the string "Hello world" that is assigned to the variable s.

The **append operator** is "**$=**" causes the string expression to the right of the operator to be appended to the end of the string currently contained in the variable to the left of the operator. For example, the statements

```
string s;
s = "abc";
s $= "def";
```

results in s having the string "abcdef".

If a numeric value is used with a string operator such as '$' the numeric value is converted to a string before the operation. For example, the statement

```
string s;
s = "ABC" $ 123;
```

results in the string "ABC123" being assigned to s.

The **substring operator** has the form "**[***start***:***length***]**" where *start* is the position of the first character in the string that is to be included in the substring. The leftmost character of a string has a position index of 0 (zero). The *length* value specifies how many characters are to be included in the substring. If you specify only the starting value then a single character is selected. If you specify a starting value and a colon but omit the length, then the substring extends from the starting character to the end of the string. The following examples show what characters are selected by various forms of the substring operator:

```
Expression           Selected substring
----------           ------------------
"abcde"[0:2]         ab
"abcde"[1:3]         bcd
"abcde"[2]           c
"abcde"[2:]          cde
```

The substring operator can be applied to variables, string constants, and string expressions. You can also apply it to non-string variables and expressions which DTL automatically converts to strings before applying the substring operator. The following are some examples:

```
Expression              Selected substring
----------              ------------------
("abc"$"123)[2:3]       c12
(121+2)[1:2]            23
```

The substring operator may also be used on the left side of an assignment statement. For example, the following statement replaces two characters in the middle of a string:

```
address[2:3] = "TN";
```

If an assignment is made to a substring that extends beyond the end of the string, the string is extended with blanks up to the start of the substring. For example, consider the following sequence of statements:

```
s = "abc";
s[4:2] = "12";
```

At the end the string variable s with have the value “abc 12”.

Assignment Operators

The following assignment operators can be used in expressions:

```
variable = expression;    // Assign expression to variable
variable += expression;   // Add expression to variable
variable -= expression;   // Subtract expression from variable
variable *= expression;   // Multiply variable by expression
variable /= expression;   // Divide variable by expression
variable $= expression;   // Append the string expression
```

Comparison Operators

The following operators compare two values and produce a value of 1 if the comparison is true, or 0 if the comparison is false:

```
==   Equal
!=   Not equal
<=   Less than or equal
>=   Greater than or equal
<    Less than
>    Greater than
```

These operators may be used with integer, real, and string values. String comparisons are done without regard to the case of the letters. For example, the expression ("ABC" == "abc") is true. If the two strings are of unequal lengths, the shorter string is extended with trailing spaces to match the length of the longer string before the comparison is done. The strcmp function performs a case-sensitive comparison.

Logical Operators

The following logical operators may be used:

```
!    Logical NOT (negates true and false)
&&   AND
||   OR
```

The result of a logical operator is integer 1 if the value is true or 0 if the value is false. For example, the statements

```
int i,j,k;
i = 1;
j = !i;
k = j || 1;
```

result in 'i' having the value 1, 'j' having the value 0, and 'k' having the value 1.

Bit Operators

The following operators perform operations on bits in an integer value.

```
~    Bitwise negation (flips value of each bit)
&    AND
|    OR
```

Conditional Operator

The conditional operator has the form:

operand1 **?** *operand2* **:** *operand3*

The value of *operand1* is evaluated. If it is true (not zero) then the value of *operand2* is the result of the expression. If the value of *operand1* is false (zero) then *operand3* is the result of the expression. For example, the expression "1?2:3" has the value 2 and "0?2:3" has the value 3.

Subscript Operator

There are two other special operators: "**[**...**]**" (square brackets) which enclose subscripts on arrays and "**,**" (comma) which is used to specify left-to-right, sequential evaluation of a list of expressions.

DTL allows you to define arrays with one or two dimensions. A subscript operator selects an individual element of an array. The first element of an array is numbered 0 (zero). So, if an array X was defined with three elements, they would be X[0], X[1], and X[2]. A two dimensional subscript is written in the form "**[***row***,***column***]**".

If both subscript and substring operators are applied to a string array, the subscript is specified first. For example, to select a substring from element 2 of an array named address, you would write it like this: address[2][3:4]. For example, consider the following statements:

```
string name[3] = {"Phil","John","Dan"};
string c1,c2,c3;
c1 = name[0][0:1];     // Gets first letter from Phil
c2 = name[1][0:1];     // Gets first letter from John
c3 = name[2][0:1];     // Gets first letter from Dan
```

This results in c1 getting the value “P”, c2 getting “J”, and c3 getting “D”.

Operator Precedence

Operator precedence, in decreasing order, is as follows: subscript, substring, unary minus, logical NOT, ++ and --, exponentiation, multiplication, division and modulo, addition and subtraction, relational (comparison), bit AND (“&”), bit OR (“|”), logical AND (“&&”), logical OR (“||”), conditional (“? ”), assignment, comma. Parentheses may be used to group terms.

Comments

The beginning of a comment is denoted with “//” (two consecutive slash characters). Everything from the “//” sequence to the end of the line is treated as a comment. Comments may be on lines by themselves or on the ends of other statements. You also can specify a comment by beginning the comment with the “/**” character sequence. All characters following this are treated as comments up to the matching “**/” sequence. This type of comment can cross lines. The following statements illustrate both types of comments:

```
//  Read the next data value
fread(f,x,y);  //  Do the read
/*
 *  Display main menu.
 */
mainmenu();                  /* This is a comment too */
```

Declarations

DTL supports three types of values: '**int**' (32-bit integer), '**double**' (64-bit floating point), and '**string**' (variable length text string). You can declare variables to hold any of these types, and you can write functions that return values of these types. Functions that do not return values are said to be of type '**void**'.

Variable types

Each variable that you use in your program must be declared before it is used. There are three types of variables: '**int**' (32-bit integer), '**double**' (64-bit floating point), and '**string**' (variable length character strings). For compatibility with the C language the keyword 'long' may be used instead of 'int'; 'real' or 'float' may be used instead of 'double'.

DTL is very "liberal" in performing type conversions. Basically, any type of variable may be used with any function or operator. When integer and real expressions are used together with arithmetic or comparison operators the integer expression is first converted to type double. When a string expression is used where a numeric value is required, the characters in the string are scanned and the string is decoded as an integer or real value as appropriate. For example, the following expressions are legal (but not necessarily good form):

```
i = 2 * "3";
s2 = sqrt("2.31");
area = 2 * X;
```

Similarly, when an integer or real expression is used where a string is expected, the value of the expression is converted to a digit string that represents the value. For example, consider the statement:

```
i = (21+3) $ "7";
```

First the integers 21 and 3 are added yielding the numeric value 24. Because the concatenation operator requires string operands the numeric value is converted to the string "24" which is concatenated with "7" yielding the string "247". Assuming 'i' is an integer variable, DTL converts the string "247" to a numeric value which is assigned to i.

When performing a conversion from a string to an integer or real value DTL stops when it encounters the first character that is not valid in an integer or real numeric value. So the statement

```
i = "23W";
```

would assign the value 23 to the integer variable i.
The statement

```
i = "ABC";
```

results in ‘i’ being set to 0.

Variable classes

There are four classes of variables: global, local, static local, and formal parameters:

global variables — These are variables that are declared outside the body of any function (usually they are defined at the top of the program before the first function). These variables may be used by any function. Global variables may be initialized when they are declared and may be assigned values from within any function. They retain their values until altered by an explicit assignment.

local variables (also known as “stack variables”) — these variables are “local” to the function in which they are defined and are only in existence while the function is active. If an initial value is specified when they are declared that value is assigned to the variable *each time* the function is entered (i.e., they do not retain the value they had when the function last exited). Two or more functions may have the same local variable names without conflict since they are separate variables. If a local variable has the same name as a global variable, the local variable is used in the function where it is defined but other functions that do not have local variables with the same name could reference the global variable.

static local variables — These variables are local to the function in the sense that their names are only meaningful to the function in which they are defined and multiple function can use static local variables with the same name without conflict. However, unlike stack local variables, static local variables are assigned permanent storage space and retain their values between function calls. If an initial value is specified when a static local variable is declared, that value is assigned to the variable once before the program begins execution. If the function with the variable is called and the value of the variable is altered, it retains the altered value on reentry to the function.

formal parameter variables — These are the variables that are listed in parentheses following the function name. At the time that the function is called DTL evaluates each of the expressions in the calling argument list and copies the values to each corresponding formal parameter. Thus on entry to the function the formal parameters have been initialized to the values that were specified as calling arguments. You are free to modify the values of the parameters during the execution of the function. When you execute a ‘return’ statement to exit from the function the values of the formal parameter variables

are copied back to the variables that were specified with the function call. This copy-back operation is only performed to calling arguments that are variables, arrays, and substring variables.

The types of variables specified as arguments to functions do not have to agree with the types of the formal parameters. If the types are different, DTL does the appropriate conversions as the values are passed into and out of the function.

The names of formal parameters must be listed in parentheses following the function name and their declarations must be either part of the list or specified between the close parenthesis and the open brace that begins the function body.

The following example shows examples of each variable class:

```
int g1;                     // Global variable
double pi = 3.14159;        // Global variable
double area(double radius) // Formal parameter
{
    double a;               // Local (stack) variable
    static int i;           // Static local variable
    a = pi * (radius^2);
    return(a);
}
```

Variable declaration statement

A variable declaration statement has the following form:

```
[class] type variable[rows,columns] = {value,value,...};
```

The first item in a variable declaration is a class keyword which may be omitted. The only class supported by DTL is ‘static’. Global variables are defined by placing the definition outside any function (it is suggested that all global variable declarations be placed at the top of the program before the first function definition).

The type of the variable is specified next. It may be ‘int’, ‘double’, or ‘string’.

The name of the variable is specified next. Variable names may be from 1 to 31 characters long. The first character must be a letter or underscore (‘_’). The following characters may be letters, digits, and underscores. Variable names are not case sensitive.

You may declare more than one variable of the same class and type with a single statement by listing the variable names, dimension sizes, and initialization values with commas

separating the variable declarations. The following are examples of variable declarations:

```
int i,j,k;
string Name;
double expdate;
static int count;
```

Array declarations

If the variable is an array, the name must be followed by the dimension size enclosed in square brackets. DTL supports arrays with one and two dimensions. For two dimensional arrays the first dimension value is the number of rows and the second value is the number of columns. The following is an example declaration for an integer array with 2 rows and 100 columns:

```
int datary[2,100];
```

The following statement declares a vector (single dimensional array) of string variables:

```
string names[20];
```

When used in expressions, the first element of an array is numbered 0, the second element 1, and so on. So, the array 'names' which is is declared to have 20 elements would be referenced using index numbers 0,1,...,19. The following is an example of a program that stores sequential numbers into an integer vector:

```
void main()
{
    int i,ary[20];

    for (i=0; i<20; i++) ary[i] = i;
}
```

Note in this example that the for loop causes 'i' to vary from 0 up to, but not including, 20 (i.e., 0 to 19).

Variable initialization

An equal sign and one or more initial values may be optionally specified following the variable name (and array dimension, if any). If the variable is not an array, simply specify an equal sign and a constant of the appropriate type. The following are examples of variable declarations with initializations:

```
int numitems = 10;
```

```
string first_name = "Phil";
static double pi = 3.14159;
```

If an array is being initialized, the list of values must be enclosed in braces and separated by commas. When initializing two dimensional arrays the initial values should be specified by rows (i.e., specify the value for row 0 column 0, row 0 column 1, row 0 column 2, etc.; the second subscript varies most quickly). The following are examples of array initializations:

```
int ivec[5] = {12, 15, 18, 21, 24};
int ary[2,3] = {00, 01, 02, 10, 11, 12};
```

If you specify fewer initialization values than the dimension size of the variable the uninitialized elements are set to zero for numeric arrays or empty strings for string arrays.

If you do not specify initialization values for a variable the variable will have a random value until you assign a value to it.

During execution of the program the initialization is handled differently depending on whether the variable is declared to be of type "static". Static variables are initialized *once* at the beginning of execution of the program. This is true even for local variables declared within a function that is called repeatedly. Local function variables that are not declared static (i.e., "automatic" stack variables) are initialized each time the function is entered. Consider the following example:

```
int g1 = 1;
void function alpha(void)
{
    int a1 = 2;
    static s1 = 3;
    <<body of function>>
}
```

The variable 'g1', which is global, and the variable 's1', which is static, are initialized once at the start of execution of the program. If they are subsequently assigned different values they retain the values until another assignment occurs. The variable 'a1' is reinitialized to the value 2 each time the alpha function is called.

Declaration examples

The following are examples of variable declarations:

```
int i,j;                 // Two integer variables
int ary[3,4];            // Integer array: 3 rows, 4 columns
double pi = 3.14159;     // Real variable with initial value
int i[2,2]={1,2,3,4};    // Integer array with values
string Name="Phil";      // String with initial value
static int first=1;      // Static int variable
```

The following is an example of a complete program with global variables, local variables, static local variables, and formal parameter variables.

```
double pi = 3.14159;              // Global variable
prototype double circlearea(1);   // Prototype for function

int main()
{
      static double radius = 5;   // Static local variable
      double area;                // Local variable

      area = circlearea(radius);
      printf("The area is %f\n",area);
      return(1);
}

double circlearea(radius)
double radius;                   // Declare formal parameter
{
      double area;               // Local variable

      area = pi * radius^2;
      return(area);
}
```

Program Statements

A program consists of *declarative* and *executable* statements. Declarative statements are used to define functions and variables; they do not perform any actions once the program is running. Executable statements define actions to be carried out by the running program. Examples of executable statements are assignments, for, while, do, and goto.

Basic Statement Syntax

The basic syntax of DTL statements is very similar to that of C programs. A DTL statement is composed of one or more "tokens" separated by spaces, tabs, and punctuation characters. Tokens consist of variable names, numeric constants, string constants, operators, and reserved keywords. Except within a string constant, space, tab, and carriage-return/line-feed character sequences are equivalent and serve as token separators.

DTL statements are terminated with a semicolon character (";"). You may freely continue a statement across multiple lines. For example, the statement:

```
y = 2 * x + sqrt(z);
```

could also be written as:

```
y = 2 *
    x +
    sqrt(z);
```

Reserved Keywords

The following keywords have reserved meanings and may not be used for the names of variables or functions (but they may be used inside quoted strings): auto, break, const, continue, do, double, else, extern, for, goto, if, int, integer, long, loop, proto, prototype, real, register, return, short, signed, static, stop, string, unsigned, void, volatile, while.

Assignment Statement

The assignment statement is an executable statement that evaluates an expression and assigns its value to a variable. The syntax for an assignment statement is:

```
variable = expression;   // Assign expression to variable
variable += expression;  // Add expression to variable
variable -= expression;  // Subtract expression from variable
variable *= expression;  // Multiply variable by expression
variable /= expression;  // Divide variable by expression
variable $= expression;  // Append expression to end of string
```

where “variable” is a variable that was previously declared. The variable may be subscripted if it is an array. “expression” is a numeric, logical, or string expression.

IF Statement

The form of the IF statement is:

```
IF (expression) statement1 [ELSE statement2]
```

If the *expression* is true (not zero) *statement1* is executed, if the expression is false (0) and the ELSE clause is specified, *statement2* is executed. The ELSE clause and the second set of controlled statements are optional. You may control groups of statements by enclosing them in braces. The following are examples of valid IF statements:

```
if (x > bigx) bigx = x;


if (firsttime) {
    for (i=0; i<100; i++) ary[i] = 0;
}
```

```
if (x < Pivot) {
    Y = B0+B1*(X-Pivot);
} else {
    Y = B0+B2*(X-Pivot);
}
```

If you have multiple conditions that you need to check for you can chain together multiple IF/ELSE statements in the following form:

```
if (expression1) {
   <<controlled statements>>
} else if (expression2) {
   <<controlled statements>>
} else if (expression3) {
   <<controlled statements>>
} else {
   <<controlled statements>>
}
```

WHILE Statement

The WHILE statement loops until the controlling expression becomes false (0) or a BREAK statement is executed within the loop. The form of the WHILE statement is:

```
while (expression) {
    << controlled statements >>
}
```

Each time around the loop the *expression* is evaluated. If it is true (non zero) the controlled statements are executed and then the process repeats until the expression becomes false. If a BREAK statement is executed within the loop, execution of the loop terminates and control is transferred to the first statement beyond the end of the loop. If a CONTINUE statement is executed in the loop, control is transferred to the conditional test at the top of the loop.

Note that the WHILE statement checks the controlling expression before executing the loop the first time so if the expression is initially false it will not execute the controlled statements at all.

The following example shows a WHILE statement that runs until we locate an asterisk character in a string.

```
index = 0;
while (s[index:1] != "*") index++;
```

Note that this example is somewhat dangerous because it does not deal with the case where the string does not contain an asterisk at all. The following sequence would be safer:

```
len = strlen(s);
index = 0;
while (index < len && s[index:1] != "*") index++;
if (index >= len) {
    printf("The string does not contain an asterisk\n");
} else {
    printf("The asterisk is at position %d\n",index);
}
```

DO Statement

The DO statement is very similar to the WHILE statement except the control expression is evaluated at the end of the loop rather than the beginning. This causes the loop always to be executed at least once. The form of the DO statement is:

```
DO {
    << controlled statements >>
} WHILE (expression);
```

For each iteration of the loop the controlled statements are executed and then the conditional expression is evaluated. If it is true (non-zero) control transfers to the first controlled statement at the top of the loop. A BREAK statement may be used to terminate the loop before the conditional expression is evaluated. A CONTINUE statement can be used to cause control to be transferred from within the loop to the point where the conditional expression is evaluated.

LOOP Statement

The LOOP statement is the simplest looping statement. The form of the LOOP statement is:

```
LOOP {
    << controlled statements >>
}
```

Note that there is no conditional expression that controls how long the loop continues. You must have an IF statement with a BREAK, RETURN, or STOP statement inside the loop to cause the loop to stop when some condition is met. A CONTINUE statement can be used in the loop to cause control to be transferred to the top of the loop. The following is an example of a LOOP statement:

FOR Statement

The FOR statement is a looping control statement similar to the WHILE statement; however, the FOR statement also allows you to specify initialization expressions that are executed once at the beginning of the loop, and loop-end expressions that are executed at the end of each loop cycle. The form of the FOR statement is:

```
FOR (expression1; expression2; expression3) statement;
```

Execution of a FOR statement proceeds as follows:

1. Evaluate *expression1*. Typically this expression will include assignment operators ("=") to set initial values for loop variables. If you need more than one initial expression, specify them as a list separated by commas.
2. Evaluate *expression2*. If its value is false (0) terminate the FOR statement and transfer control to the statement that follows the controlled statement. If *expression2* is true, proceed to the next step.
3. Execute the *controlled statement*. If more than one statement is to be controlled, enclose them with brace characters ("{" "}").
4. Evaluate *expression3*. This expression will typically contain operators such as "++", "+=", "--", or "-=" to modify the value of a loop variable.
5. Transfer control to step 2, where *expression2* is once again evaluated.

The following is an example of a FOR statement:

```
sum = 0;
for (i=0; i<10; i++) sum += a[i];
```

Frequently you will want to have more than a single statement controlled by the FOR statement. To do this enclose the controlled statements in braces. For example, the following FOR statement controls 2 statements:

```
sumx = 0;
sumx2 = 0;
for (i=0; i<10; i++) {
   sumx += x[i];
   sumx2 += x[i]^2;
}
```

It is possible to specify multiple expressions for ‘expression1’ and ‘expression3’. To do this, separate the subexpressions with commas and they will be executed in order. For example, the following statement initializes ‘i’, ‘j’, and ‘sum’ to zero before the loop is started and increments both ‘i’ and ‘j’ at the end of the loop:

```
for (i=0,j=0,sum=0; i<10; i++,j++) {
    << controlled statements >>
}
```

BREAK Statement

The BREAK statement can be used in FOR, WHILE, LOOP, and DO loops to terminate the loop and cause control to transfer to the statement beyond the end of the loop. The following is an example of a BREAK statement:

```
time = 0;
x = 0;
while (time < endtime) {
    x += delta * xspeed;
    if (x > 10) break;
}
```

CONTINUE Statement

The CONTINUE statement can be used in FOR, WHILE, LOOP, and DO loops to terminate the current iteration and begin the next one. When CONTINUE is executed in a WHILE or DO statement, control is transferred to the point in the loop where the loop control expression is evaluated. When CONTINUE is executed in a FOR statement, control is transferred to the bottom of the loop where expression3 is evaluated (which normally augments the values of the loop variables for the next iteration). The following is an example of a CONTINUE statement that skips over the statements that follow it and advances to the next value of 'i' if x[i] is negative.

```
sumsqrt = 0;
for (i=0; i<10; i++) {
   if (x[i] < 0) continue;
   sumsqrt += sqrt(x[i]);
}
```

GOTO Statement

The GOTO statement causes program control to be transferred to a statement with a specified label. The form of the GOTO statement is:

```
GOTO label;
```

where "*label*" is a label on the statement to which control is to be transferred. The following is an example of a GOTO statement:

```
int main()
{
    string s,c;
    int i;

    s = "ABC*DEF";
    i = 0;
top:    c = s[i:1];
    if (c == "*") goto end;
    i++;
    goto top;
end:    printf("* is at location %d\n",i);
    return(1);
}
```

Note that it is usually better programming form to use FOR, WHILE, DO, or LOOP statements to perform looping type operations rather than GOTO.

RETURN Statement

The RETURN statement is used to return control from a function to the calling program or function. Executing a RETURN statement in the main program terminates the entire DTL program.

The form of the statement is:

```
RETURN [expression];
```

If the function being executed returns a value (i.e., it is not of type void) then you must specify an *expression* with the RETURN statement. This expression is the value returned for the function. If the function is of type void then the RETURN statement should not have an associated expression. The following is an example of a RETURN statement:

```
prototype double circlearea(1);

int main()
{
    double radius,area;

    radius = 2;
    area = circlearea(radius);
    printf("Radius = %f, Area = %f\n",radius,area);
    return(1);
}

double circlearea(radius)
{
    double area;

    area = 3.14159 * radius ^ 2;
    return(area);
}
```

Functions

The only practical way to write a large and complex program is to divide into manageable units. Ideally, this division should be done so that each unit performs a well defined task and the interdependencies between units is minimized. The “function” is the basic program unit in the DTL programming language (other programming languages use the term “procedure” or “subroutine” for essentially the same thing).

In DTL there are two classes of functions: (1) built-in library functions that are written by the developers of DTL and are available for your use, and (2) user defined functions that you write and call from your DTL programs. Many built-in library functions are provided with DTL. This chapter is focused toward user defined functions.

Before getting into the details of using functions let’s look at a simple example that has a main program and a function. The function accepts a string argument and returns an index number indicating the character position of the first ‘*’ character in the string:

```
prototype int findstar(1);
int main()
{
    int index;
    index = findstar("ABC*DEF");
    printf("The first * is at offset %d\n",index);
    return(1);
}

int findstar(string s)
{
    int i,len;
/*
 *  Determine the length of the string.
 */
    len = strlen(s);
/*
 *  Try to find the first occurrence of '*'.
 */
    for (i=0; i<len; i++) {
        if (s[i:1] == "*") return(i);
    }
/*
 *  String does not contain a '*'.  Return -1.
 */
    return(-1);
}
```

Declaring Functions

The general form of a function declaration is as follows:

```
[type] name([formal parameters])
[parameter declarations]
{
      body of function
}
```

Here is an example of a simple function that accepts two integer arguments, adds them together and returns the sum as the result of the function:

```
int add(value1,value2)
int value1;         // First value to add
int value2;         // Second value to add
{
       int sum;

       sum = value1 + value;
       return(sum);
}
```

The first item in a function declaration is the type keyword that specifies what type of value the function will return. The valid keywords are "int", "double", "string", and "void". If you do not specify a keyword, void is assumed by default. Void means that the function does not return a value.

The name of the function must follow the type keyword (if there is a type specification). Function names have the same form as variable names: they must begin with a letter or an underscore and following the first character may consist of letters, digits, and underscores. They may be up to 31 characters long and they are not case sensitive.

The name of the function must be followed by the list of "formal parameters" for the function enclosed in parentheses. If there are no formal parameters specify "()" or "(void)" after the function name.

Formal parameters are like local variables for the function and may have the same names as formal parameters or local variables for other functions. You may not define a local variable with the same name as a formal parameter. When a function is called the value of the expression that corresponds positionally to the formal parameter is copied into the formal parameter. When the function exits the value of the formal parameter at the time that the function exits is copied back to the calling argument. The copy-back is done if, and only if, the calling argument is a variable (possibly subscripted or substringed).

Array arguments are handled differently. Rather than copying all of the values for an array into the local variable for the function, DTL indirectly references the array so that as you access or alter the formal parameter variable in the function the values in the array that was passed as an argument to the function are accessed.

You can call a function using arguments that have different types than the corresponding formal parameters. In this case DTL converts the values to the correct type for the formal parameters on entry to the function and converts the value of the formal parameters on exit to the type of the argument variables as the values are passed back.

There are two ways to specify the types of the formal parameters. The first method is to specify a type keyword in front of each parameter name in the argument list. The following example illustrates this type of declaration:

```
int sub1(int i, double r, string s)
{
     << body of function >>
}
```

The second method is to declare the types of the parameters using separate statements between the close parenthesis at the end of the parameter list and the open brace that begins the function body. The following example illustrates this type of declaration:

```
int sub1(i,r,s)
int i;            // Integer formal parameter
double r;         // Real formal parameter
string s;         // String formal parameter
{
     << body of function >>
}
```

The formal parameter declarations use the same style as variable declarations. Note that there is no semicolon between the close parenthesis at the end of the formal parameter list and the first parameter declaration or the open brace for the function body.

You can choose which style of parameter type declaration to use based on your own taste. There is no difference in terms of the effect on the program.

Array parameters

If a formal parameter is an array, the number of rows and columns is determined by the size of the array that is specified as the argument when the function is called. In other words, a

formal parameter array “conforms” to the size of whatever array is passed to an argument so you can write a function that will work with different size arrays. Because of this, the number of columns and rows specified for the formal parameter array are ignored and you may specify 0 to emphasize that the size is dependent on the passed array. The arraysize1() and arraysize2() library functions can be used to determine the actual number of rows and columns that an array has.

The following is an example of a program with a function that accepts a single dimension array and returns the sum of the elements in the array:

```
prototype int vsum(1);

int main()
{
    int vec[5] = {1,2,3,4,5};
    int sum;

    sum = vsum(vec);
    printf("The sum is %d\n",sum);
    return(1);
}

/*
 *  Function to compute sum of entries in an array.
 */
int vsum(vec)
int vec[0];
{
    int i,sum,size;

/*
 *  Use the arraysize1 library function to determine number of
 *  items in the array.
 */
    size = arraysize1(vec);
/*
 *  Compute sum of entries.
 */
    sum = 0;
    for (i=0; i<size; i++) {
        sum += vec[i];
    }
    return(sum);
}
```

Function Prototypes

A function prototype statement provides information to the DTL compiler about a function whose definition has not yet been encountered by the compiler. You must specify a prototype for any function whose return type is not void if you have a call of the function before the function definition.

The form of a function prototype statement is:

```
prototype type function(number_of_parameters);
```

Where '*type*' specifies the type of value returned by the function. it must be 'int', 'double', 'string', or 'void'. 'Void' indicates that the function does not return a value. After the function name specify in parentheses the number of parameters that the function expects. Note to C programmers: since DTL performs automatic run-time conversion of argument types it is not necessary to specify the type of the formal parameters, only a count of how many there are.

Function prototype declarations must occur before the point where a function is called. Typically they are grouped together before the first function and frequently they are stored in an external file that is included in the source program by use of the #include statement.

The following are examples of function prototype statements:

```
prototype double area(1);
prototype void showmenu(0);
prototype int countusers(2);
```

Invoking Functions

To invoke a function simply specify the name of the function followed by an open parenthesis, any argument values, and a close parenthesis. If the function does not require any arguments specify "()" after the function name.

DTL functions may call themselves recursively. The following example shows a function that computes a factorial by recursive calls to a function:

```
prototype int fact(1);

int main()
{
    int f;

/*
 *  Compute and print 5 factorial.
 */
    f = fact(5);
    printf("5 factorial = %d\n",f);
    return(1);
}

/*
 *  Function to recursively compute a factorial.
 */
int fact(value)
{
    int f;

/*
 *  One factorial is 1.
 */
    if (value == 1) return(1);
/*
 *  Call this routine recursively to compute the factorial.
 */
    f = value * fact(value-1);
    return(f);
}
```

Built-In Library Functions

The functions described in the following chapters are predefined and built into DTL. The prototypes are also built in so you do not need to specify function prototypes.

Each function description that follows includes a prototype specification for the function. The prototype shows the type of value returned by the function, the number of arguments, and the type of each argument. The special type designation “any_type” means that any type argument may be specified. If you specify an argument that is of a different type than shown by the prototype, the type of the argument is automatically converted to the correct type when the function is called.

For example, here is the prototype for the strlen function:

```
int strlen(string str)
```

This prototype indicates that the strlen function returns an int type value and it expects a single string argument.

Some functions accept a variable number of arguments. In this case the optional arguments are enclosed in bracket characters in the prototype specification. For example, the following is the prototype for the fopen function:

```
int fopen(filename,mode[,share][,recsize])
string filename;   /* File specification */
string mode;       /* File open mode */
string share;      /* (optional) File sharing flags */
int recsize;       /* (optional) Relative file rec size */
```

The fopen returns an int type value. It requires at least two string arguments (‘filename’ and ‘mode’). There is an optional third string argument named ‘share’.

Function Error Status

Many library functions may encounter errors under some conditions. For example, if you use the fopen() function to attempt to open an existing file and the file does not exist, an error is detected. When an error occurs an error code is stored in an internal library data cell. This error code can be retrieved by subsequent use of the lasterror() function. The errormsg() function converts an error status code into a text message describing the error. The perror() function can be used to display a message associated with an error code.

String Functions

String variables and functions are an important part of the DTL programming language. String variables have variable lengths, they can hold different length strings at different times. DTL strings can hold binary data including the null character.

strcmp — String comparison

Function prototype:

```
int strcmp(string str1, string str2)
```

The strcmp function performs a case sensitive comparison between two strings and returns the following result values:

```
 1 = str1 is greater than str2
 0 = str1 is equal to str2
-1 = str1 is less than str2
```

This function is similar to the "==" relational comparison operator except the "==" operator is case insensitive whereas the strcmp function is case sensitive. A second difference is that when the "==" operator is used to compare two strings of unequal lengths, the shorter string is extended with blanks to match the length of the longer string. The strcmp function does not extend shorter strings. For example, the expression "ABC"=="ABC " is true because the first string is extended with blanks to match the length of the second string. On the other hand, strcmp("ABC","ABC ") returns the value –1 because the first string is less than the second string.

strlen — Determine length of string

Function prototype:

```
int strlen(string str)
```

This function returns an integer value corresponding to the number of characters in the string argument. For example, the value of strlen(“abc”) is 3. Some functions return null (empty) strings in some cases. The strlen function is the easiest way to check for a null length string.

space — Create blank filled string

Function prototype:

```
string space(int length)
```

The space function creates a string of space characters of the specified length.

trim — Remove spaces from end of a string

Function prototype:

```
string trim(string instr)
```

The trim function accepts a string argument, removes any trailing spaces from the end of the string, and returns the resulting string as its result.

cleanspaces — Clean up spaces in string

Function prototype:

```
string cleanspaces(string instr)
```

The cleanspaces function removes extraneous spaces and tabs from a string. It operates in the following steps: (1) convert any tab characters in the string to spaces; (2) remove any spaces from the front of the string; (3) remove any spaces from the end of the string; (4) collapse multiple consecutive spaces within the string to a single space.

repeat — Create string with repeated pattern

Function Prototype:

```
string repeat(string instr, int count)
```

The repeat function creates a string with a specified pattern string repeated a specified number of times. For example, the function call repeat(“abc”,3) produces the string “abcabcabc”.

locate — Locate substring in string

Function prototype:

```
int locate(string substr, string primary, int startpos)
```

The locate function tries to find the first occurrence of a substring specified by ‘substr’ in a string specified by ‘primary’. The target substring may consist of one or more characters. The comparison is case sensitive. f the substring is found the value returned by the function is the index number of the character in the primary string where the first character of the substring occurs. The index number is 0 based (i.e., if the substring starts at the beginning of the primary string the value will be 0). If the substring is not found in the primary string –1 is returned. The comparison is case sensitive.

If the optional third argument, ‘startpos’, is specified the search begins at the specified starting character position in the primary string. If you do not specify the ‘startpos’ argument a value of 0 is used by default and the search starts from the beginning of the primary string.

The following are example function calls and the resulting values:

```
locate("xy","abcxydefxy")      =   3
locate("xy","abcxydefxy",4)    =   8
locate("rx","abcxydefxy")      =  -1
```

rlocate — Reverse locate substring in string

Function prototype:

```
int rlocate(string substr, string primary, int startpos)
```

The rlocate function tries to find the last occurrence of a substring specified by ‘substr’ in a string specified by ‘primary’. The comparison is case sensitive. If the substring is found the value returned by the function is the index number of the character in the primary string where the first character of the substring occurs. The index number is 0 based (i.e., if the substring starts at the beginning of the primary string the value will be 0). If the substring is not found in the primary string –1 is returned. The target substring may consist of one or more characters. The comparison is case sensitive.

If the optional third argument, ‘startpos’, is specified the search begins at the specified starting character position in the primary string. If you do not specify the ‘startpos’ argument the search begins at the right end of the primary string and progresses to the left.

The following are example function calls and the resulting values:

```
rlocate("xy","abcxydefxy")      =   8
rlocate("xy","abcxydefxy",6)    =   3
rlocate("rx","abcxydefxy")      =  -1
```

strcount — Count occurrences of a substring

Function prototype:

```
int strcount(string substr, string primary)
```

The strcount function counts the number of occurrences of a substring specified by the ‘substr’ that occurs in the string specified by the ‘primary’ argument. The target substring may consist of one or more characters. The comparison is case sensitive. For example, the following call would return a value of 2.

```
count = strcount("A","ABCADE");
```

This function can be used to determine how many lines will be required to print a string with embedded line-feed characters. To do this, specify “\n” as the substing to search for.

strupr — Convert string to upper case

Function prototype:

```
string strupr(string instr)
```

The strupr function converts all lower case letters in the input string to upper case and returns the resulting string as the result. Characters in the input string that are not lower case letters are not changed. For example, strupr(“Abc12”) produces the string “ABC12”.

strlwr — Convert string to lower case

Function prototype:

```
string strlwr(string instr)
```

The strlwr function converts all upper case letters in the input string to lower case and returns the resulting string. Characters in the input string that are not upper case letters are not changed. For example, strlwr(“Abc12”) produces the string “abc12”.

mixcase — Convert string to mixed case

Function prototype:

```
string mixcase(string instr)
```

Converts a string to mixed upper and lower case. The first character of the string is capitalized and also letters following space or period. Other letters are converted to lower case. For example, the value of mixcase(“PHIL sherrod”) is “Phil Sherrod”.

translate — Translate characters in string

Function prototype:

```
string translate(string str1, string str2, string str3)
```

The translate function translates the characters in ‘str1’ according to the two strings ‘str2’ and ‘str3’. Each character in ‘str1’ is examined. If it matches any character in ‘str2’ the character in ‘str3’ which is at the same position in the string as the matching character in ‘str2’ is substituted for the original character in ‘str1’. Any character in ‘str1’ that is not found in ‘str2’ is left untranslated. The character comparison is case sensitive. The returned value of the function is the translated string. The input arguments are not altered. For example, the statements

```
string s;
s = translate("Alpha1","Aa","Xs");
```

result in ‘s’ receiving the translated string “Xlphs1”.

char — Convert ASCII value to character

Function prototype:

```
string char(int value)
```

The char function produces a single character whose ASCII code corresponds to the integer value argument. For example char(65) produces the letter ‘A’ and char(66) produces the letter ‘B’.

ichar — Convert character to ASCII value

Function prototype:

```
int ichar(string c)
```

The ichar function returns an integer value that corresponds to the ASCII code for the character that is specified as an argument. If a string is provided as an argument, the ASCII value of the first character is returned. For example, ichar(“A”) is 65.

is*xxxx* — Character type tests

Function prototypes:

```
int isxxxxx(string c)
```

DTL provides a set of library functions for testing to see if a character is part of a particular set. For example, the isalpha function returns the value true (1) if the specified string consists only of upper or lower case letters and false (0) if the string contains any characters other than upper and lower case letters. Each of the following functions accepts a string argument and returns an integer value that is either 0 or 1.

int isalpha(c) — Tests if the string consists of upper or lower case letters.

int islower(c) — Tests if the string consists of lower case letters.

int isupper(c) — Tests if the string consists of upper case letters.

int isdigit(c) — Tests if the string consists of digits.

int isalnum(c) — Tests if the string consists of letters or digits.

int iscntrl(c) — Tests if the string consists of control characters (i.e., characters whose ASCII code is less than hex 20).

int isgraph(c) — Tests for printable characters not including space (i.e., characters whose ASCII code is in the range hex 21 to 7E).

int isprint(c) — Tests for printable characters (i.e., characters whose ASCII code is in the range hex 20 to 7E).

int ispunct(c) — Tests for punctuation characters.

int isspace(c) — Tests for space, form-feed, carriage-return, line-feed, tab, and vertical tab.

int isxdigit(c) — Tests for hexadecimal digits (‘0’-’9’, ‘A’-’F’, or ‘a’-’f’).

For example:

```
isalpha("a")     ==>   1 (true)
isalpha("1")     ==>   0 (false)
isalpha("a1")    ==>   0 (false)
isdigit("123")   ==>   1 (true)
isdigit("1A3")   ==>   0 (false)
```

insert — Insert one string in another

Function prototype:

```
string insert(string str1, string str2, int pos)
```

The insert function returns a string corresponding to ‘str1’ with ‘str2’ inserted in front of character position ‘pos’ of ‘str1’. A value of 0 for ‘pos’ would insert ‘str2’ at the front of ‘str1’. For example, insert(“abcdef”,“12”,2) produces “ab12cdef”.

element — Locate substring using delimiters

Function prototype:

```
string element(int number, string delim, string main)
```

The element function extracts a substring delimited by ‘delim’ characters from the ‘main’ string. The ‘number’ argument specifies which occurrence of the substring if there is more than one. The first occurrence is number 0. The following examples illustrate its action:

```
element(0,".","ab.cd.ef")      ab
element(1,".","ab.cd.ef")      cd
element(2,".","ab.cd.ef")      ef
element(3,".","ab.cd.ef")      (null string)
```

validate — Check validity of characters

Function prototype:

```
int validate(string test, string valchar)
```

The validate function searches the ‘test’ string to see if it contains any characters that are *not* contained in the ‘valchar’ string. If there are any characters in ‘test’ that are not part of ‘valchar’ it returns the position number of first such character found in ‘test’. The index is 0 based. If all of the characters in ‘test’ are contained in ‘valchar’, it returns -1. For example, the value of validate(“abxba”,“abcd”) is 2 because character in position 2 is ‘x’ which is not contained in “abcd”.

strip — Remove characters from a string

Function prototype:

```
string strip(string source, string remchar)
```

The strip function returns a string consisting of the ‘source’ input string with any characters that are found in the ‘remchar’ string removed. For example, the value of

strip(“abcdef”,“acf”) is “bde”. The strclean function performs the reverse operation.

strclean — Remove all but specified characters

Function prototype:

```
string strclean(string source, string wantchars)
```

The strclean function returns a string consisting of the ‘source’ input string with all characters removed *except* the characters specified in the ‘wantchars’ string. For example, the value of strclean(“(615)-327-3670”,“0123456789”) is “6153273670” because all characters other than digits were removed. The strip function performs the reverse operation.

Math Functions

DTL includes a generous assortment of math functions that can be used for scientific, statistical, or business calculations. All of the math functions use 64-bit floating point values and produce values accurate to about 18 significant digits.

abs — Absolute value

Function prototype:

```
double abs(double value)
```

Computes the absolute value of the argument.

acos — Arc cosine

Function prototype:

```
double acos(double value);
```

Computes the arc cosine of the value. The returned value is an angle in radians..

asin — Arc sine

Function prototype:

```
double asin(double value)
```

Computes the arc sine of the value. The returned value is an angle in radians..

atan — Arc tangent

Function prototype:

```
double atan(double value)
```

Computes the arc tangent of the value. The returned value is an angle in radians..

ceil — Ceiling

Function prototype:

```
double ceil(double x)
```

Computes the ceiling of x. Returns the smallest integer that is at least as large as x. For example, ceil(1.5)=2; ceil(4)=4; ceil(-2.6)=-2.

cos — Cosine

Function prototype:

```
double cos(double angle)
```

Computes the cosine of the angle. The angle must be in radians.

cosh — Hyperbolic cosine

Function prototype:

```
double cosh(double x)
```

Computes the hyperbolic cosine of x.

cot — Cotangent

Function prototype:

```
double cot(double angle)
```

Computes the cotangent of the angle. The angle must be in radians.

csc — Cosecant

Function prototype:

```
double csc(double angle)
```

Computes the cosecant of the angle. The angle must be in radians.

deg — Convert radians to degrees

Function prototype:

```
double deg(double angle)
```

Convert an angle that is in units of radians to the equivalent angle in units of degrees.

exp — Exponential

Function prototype:

```
double exp(double x)
```

Compute the value of e (the base of natural logarithms) raised to the x power.

fabs — Absolute value

Function prototype:

```
double fabs(double x)
```

Compute the absolute value of x. This function is equivalent to abs and is provided for compatibility with C.

factorial — Factorial

Function prototype:

```
double factorial(double x)
```

Compute x factorial (x!). Note, the factorial function is computed using the gamma function (factorial(x)=gamma(x+1)) so non-integer argument values may be computed.

floor — Floor

Function prototype:

```
double floor(double x)
```

Returns the largest integer that is less than or equal to x. For example, floor(2.5)=2; floor(4)=4; floor(-3.6)=-4.

log — Natural logarithm

Function prototype:

```
double log(double x)
```

Computes the natural logarithm (base e) of x.

log10 — Base 10 logarithm

Function prototype:

```
double log10(double x)
```

Computes the logarithm base 10 of x.

max — Maximum value

Function prototype:

```
double max(double val1, double val2, double val3,...)
```

Returns the value of the largest argument. Up to 40 argument values may be specified.

min — Minimum value

Function prototype:

```
double min(double val1, double val2, double val3,...)
```

Returns the value of the smallest argument. Up to 40 argument values may be specified.

npd — Normal probability distribution

Function prototype:

```
double npd(x, mean, std)
double x;      /* Point on std distribution curve */
double mean;   /* Mean value of distribution */
double std;    /* Standard dev of distribution */
```

Normal probability distribution of x with specified mean and standard deviation. X is in units of standard deviations from the mean.

rad — Convert degrees to radians

Function prototype:

```
double rad(double angle)
```

Convert an angle in units of degrees to the equivalent angle in units of radians.

random — Random number

Function prototype:

```
double random()
```

Returns a random value uniformly distributed in the range 0 to 1.

round — Round to integer

Function prototype:

```
double round(double x)
```

Rounds x to the nearest integer. For example, round(1.1)=1; round(1.8)=2; round(-2.8)=-3;

sec — Secant

Function prototype:

```
double sec(double angle)
```

Computes the secant of the angle. The angle must be in radians.

Secant of x. (sec(x) = 1/cos(x)).

sin — Sine

Function prototype:

```
double sin(double angle)
```

Computes the sine of the angle. The angle must be in radians.

sinh — Hyperbolic sine

Function prototype:

```
double sinh(double x)
```

Compute the hyperbolic sine of an angle.

sqrt — Square root

Function prototype:

```
double sqrt(double x)
```

Square root of x.

tan — Tangent

Function prototype:

```
double tan(double angle)
```

Computes the tangent of the angle. The angle must be in radians.

tanh — Hyperbolic tangent

Function prototype:

```
double tanh(double x)
```

Compute the hyperbolic tangent of x.

Array Functions

DTL supports one dimensional arrays (vectors) and two dimensional arrays (matrices). Array entries may hold integer, real, and string values; string values are of variable lengths.

As with scalar variables, arrays may be global (defined outside any function) or local (defined within a function). Local arrays may be static, in which case they hold their values between function calls, or dynamic, in which case they are freed and reallocated each time the function is called.

Arrays may be passed as arguments to functions. In this case references to the array from within the function are mapped to the actual array that is passed as the argument to the function. The size of the array within the function is set to match the size of the array that is passed as an argument; the size is said to "conform" to the size of the passed array. This makes it possible to write general purpose array manipulation routines that can be called at different times with different size arrays. The arraysize1() and arraysize2() functions that are described later in this chapter are used to determine the actual size of an array.

In addition to function array parameters that conform to the size of the passed array, DTL also lets you change the size of static and global arrays. This is a very powerful feature because it allows you to write programs that can handle varying amounts of data without having to declare all arrays with the maximum possible sizes. The resize function which is described below is used to change the size of an array.

resize — Change the size of an array

Function prototype:

```
void resize(array, size1 [,size2])
any_type array;    /* Name of array to be resized */
int size1;         /* New size for first dimension */
int size2;         /* New size for second dimension */
```

The resize function alters the size of an array. The '*array*' argument is the name of the array whose size is to be changed. The array may be of any type and may have one or two dimensions. The '*size1*' argument is the new size for the first dimension (or only dimension if it is a one dimensional array). The '*size2*' argument is the new size of the second dimension; it should be specified if and only if the array has two dimensions.

If you reduce the size of an array the old elements that are no longer present are discarded and freed. If you increase the size of an array the values of the original elements are preserved and the new elements are set to 0 for int and real arrays or empty strings for string type arrays.

If you use the resize function to change the size of an array that has been passed as an argument to the function, the size of the array that was passed in the call is changed. If you change the size of a global array or a local static array the size change remains in effect until it is changed again by calling resize. If you change the size of a dynamic local array (i.e., a stack array) the size change remains in effect only until the function exits; the array is reallocated with its declared size on the next call of the function.

The following example opens a file and reads each line of the file into a string array named filelines. The size of the filelines array is increased using the resize function to hold each additional line that is read. Note: although this technique works and is very efficient in terms of memory space utilization is it extremely inefficient in terms of speed. Each time resize is called it allocates a new array, copies the existing elements from the old array to the new one, and then deallocates the old array; this is an expensive operation. A more efficient approach would be to expand the array in large steps (say 1000 entries) each time it fills up.

```
int main()
{
    int f,line,status;
    string inline,filelines[1];

/*
 *  Open the file.
 */
    f = fopen("test.dat","rt");
    if (f == 0) {
        print("Unable to open file");
        return(0);
    }
/*
 *  Begin loop to read from the file.
 */
    for (line=0; ; line++) {
/*
 *  Read the next line.  Exit loop if end of file hit.
 */
        /* Try to read another line from the file */
        status = fread(f,inline);
        /* Exit loop if read returned an error code */
        if (status != 0) break;
/*
 *  Expand the size of the array to hold this line.
 */
        if (line >= arraysize1(filelines)) {
            /* Expand the array */
```

```
            resize(filelines,line+1);
        }
/*
 *  Store the line into the array.
 */
        filelines[line] = inline;
    }
/*
 * Close the file.
 */
    fclose(f);
/*
 *  Display each line in the array.
 */
    for (line=0; line<arraysize1(filelines); line++) {
        print(filelines[line]);
    }
    return(0);
}
```

arraysize — Determining size of an array

Function prototypes:

```
int arraysize1(array)
any_type array;    /* Name of array */

int arraysize2(array)
any_type array;    /* Name of array */
```

The arraysize1 and arraysize2 functions determine the size (number of elements) of the first and second dimensions of an array. Arraysize2 should only be applied to an array with two dimensions. These functions are especially useful inside functions to determine the size of the array passed as an argument to the function. However, they can be used to determine the size of any array.

The following example shows a function named arraysum that accepts a two dimensional array of arbitrary size and returns the sum of its elements:

```
prototype int arraysum(1);
void main()
{
    static a[2,3] = {4, 1, 5, 2, 7, 3};
    int sum;

    sum = arraysum(a);
    print("The sum is",sum);
    stop;
}

/*
 *  Function to sum the elements of an array.
 */
int arraysum(x)
int x[0,0];   /* Array to be summed */
{
    int sum,i,j;

/*
 *  Sum the entries in the array.
 */
    sum = 0;
    for (i=0; i<arraysize1(x); i++) {
        for (j=0; j<arraysize2(x); j++) {
            sum += x[i,j];
        }
    }
/*
 *  Finished -- Return the sum.
 */
    return(sum);
}
```

sort — Sort an array

Function prototype:

```
void sort(keyvec, indexvec [,numitems]);
any_type keyvec;     /* Vector with keys to sort */
int indexvec;        /* Parallel index vector */
int numitems;        /* (optional) number of items to sort */
```

The sort function sorts an array of items. The keys to be sorted may be of any type: integer, real, or string. String sorts are done ignoring case differences.

Rather than rearranging the items in the array being sorted, the sort function produces an index array that has the subscripts for the key array sorted in ascending order. For example, consider the following statements:

```
int keys[4] = {3,1,4,2};
int index[4];
sort(keys,index);
```

The sort function examines the values in the keys array and stores into the index array the subscripts for the keys array that would cause the key values to be in ascending order. In the case of this example, the values stored in the index array will be 1, 3, 0, 2 (remember, array subscripts are 0 based). Thus keys[1] has the smallest value (1), keys[3] has the next smallest value (2), etc. To access the elements of the key in ascending order, use the values in successive elements of the index array as subscripts. That is, keys[index[0]] is the smallest entry, keys[index[1]] is the next smallest, etc. If you wish to access the elements of the key in descending order, use the sorted index array elements in reverse order.

The third argument to sort, ‘numitems’, is optional. If specified, it is the number of items in the ‘keyvec’ and ‘indexvec’ arrays to be sorted; this may be less than the dimension size of the key and index arrays. If you omit the ‘numitems’ argument then the sort function sorts all elements in the key vector and expects the index vector to be at least as large as the key vector.

The following example is a complete program which sorts a small array and prints the entries in sorted order. Note how the values in the index array are used as subscripts to the names array to cause the names to be printed in ascending order.

```
int main()
{
    static string names[8] = {"Phil","John","Dan",
                              "Harry","Richard","Vicki",
                              "Steve","Debi"};
    int index[8],i;
    sort(names,index,8);
    for (i=0; i<8; i++) {
        print(names[index[i]]);
    }
    return(1);
}
```

Lag Functions

The functions described in this chapter are used to access prior values of variables. The lag() function is often used when processing time series data to reference an earlier value in the series.

lag — Get previous value of variable or expression

Function prototype:

```
double lag(double expression, int index)
```

Each time the lag() function is called, it stores the specified value of the *expression* argument and retuns some previously stored expression value. The *index* argument selects which previous value is to be returned. If the value of *index* is 1 then the previous value is returned, if *index* is 2 than the value from two calls earlier is returned.

The *expression* argument can be a simple variable name or a complex expression.

The *index* argument should have a constant value.

Often, the lag() function is used in DTL programs that process time series data. The values stored by lag() are global and retained between each iteration of the program invocation, so lag() can be used to reference a previous value of a variable.

If the lag() function requests a value prior to a stored value, the missing value is returned. For example, the first time lag(x,1) is called, there is no previous value of x, so missing value is returned.

Input/Output Functions

The functions described in this chapter perform general I/O operations.

Functions such as print and printf which in a C program would write output to the console, write the output to the DTREG run log file.

print — Print a line of values

Function prototype:

```
int print(val1,val2,...)
any_type val1;     /* First value to print */
any_type val2;     /* Second value to print */
```

The print function writes to the log file a line of output consisting of each of the values concatenated with a single space separating them. The cursor is advanced to the next line after the values are printed. The returned value of the function is 0 if the function is successful. An error code is returned if an error occurs during the function execution.

A variable number of arguments may be specified. Integer values are converted to character strings before they are displayed. For example, the function print("Beginning execution") would display the string "Beginning execution" at the current cursor location and then would advance the cursor to the start of the next line. Here is an example:

```
i = 123;
print("The value of i is",i);
```

This would display the line "The value of i is 123".

printf — Formatted print function

Function prototype:

```
int printf(format,val1,val2,...)
string format;   /* Format pattern string */
any_type val1;   /* First value to insert */
any_type val2;   /* Second value to insert */
```

The printf function is an extremely versatile way for your program to write messages to the execution log. Printf requires at least one argument, the 'format' string which consists of ordinary characters, control characters such as "\n" which prints a carriage-return and line-feed sequence, and conversion specifications such as "%d" and "%s" which cause successive value arguments ('val1', 'val2', etc.) to be converted to printable form and inserted in the string that is displayed.

The simplest form of the printf function has only a single argument, the format string. In this case the string is simply written to the log file. For example, the following statement displays the string "Program is running",

```
printf("Program is running\n");
```

You can print multiple lines with a single statement by embedding "\n" in the string where ever you want to advance to a new line. For example, the statement

```
printf("line 1\nline 2\nline 3\n");
```

prints three lines:

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

If you wish to have values inserted in the string as it is printed you can place conversion operators in the format string. Conversion operators begin with a percent sign ('%') and end with a conversion character. Between the percent sign and the conversion character you may have the following optional items:

- A **minus sign**, which specifies that the converted value is to be left justified in the

field.

- A **digit string** specifying a minimum field width. The converted value will use at least this many columns and more if necessary. If the converted value requires fewer characters than then specified width it will be padded on the left (or right if minus sign was specified) to fill the field. The padding character is blank normally but zero will be used if you specify a 0 as the first digit of the field width value. For example, the specification "%6d" would print a 6 digit numeric value with leading blanks if necessary to fill the field. The specification "%06d" would also print a 6 digit numeric field but leading zeros would be used to fill the field if necessary.
- A **period** followed by a number which specifies the maximum number of digits to be printed to the right of the decimal point for real values or the number of characters to be printed from a string value. For example, the specification "%8.4f" prints a real value in a field that is 8 characters wide and has 4 characters to the right of the decimal point.

The conversion character comes after the percent sign and the optional items described above. The following conversion characters may be used:

s — Insert a string value into the format string.

d — An integer value is converted to a decimal digit string. A leading minus sign is printed if the value is negative. This conversion character may be used for int items or real items if you do not want any decimal places printed. For example, the specification "%d" prints an integer value using the minimum number of characters required to display the value.

f — The value is converted to a real digit string with a decimal point and decimal digits. See the optional items described above for information about specifying the maximum field width and the number of digits to the right of the decimal point. For example, "%f" prints a real value using the minimum number of characters required to display the value. "%8.4f" prints a real value using a total of 8 characters for the field and printing 4 digits to the right of the decimal point.

e — The value is printed in scientific notation of the form "-*m.nnnn*e+*xx*".

E — Same as %e except the letter 'E' in the printed string is upper case.

g — The "%g" conversion operator is equivalent to using %e or %f, whichever is shorter. In other words, the printed value will either be in natural form (*nnn.nnn*) or scientific notation (*n.nnnn*e+*xx*) depending on the range of the value being printed.

G — Same as %g except the letter 'E' used for scientific notation format is upper case.

x — Print the value as a hexadecimal digit string in the form "0x*nnnn*".

Here are some example statements showing what they print:

```
int i = 5;
double x = 3.14;
string s = "Phil";

printf("i is %d\n",i);                i is 5
printf("x is %f\n",x);                x is 3.14
printf("s is %s\n",s);                s is Phil
printf("%s has %d dollars\n",s,i);    Phil has 5 dollars
printf("x = %10.3E\n",x);             x =  3.140E+00
```

format — Format value string

Function prototype:

```
string format(format,val1,val2,...)
string format;   /* Format pattern string */
any_type val1;   /* First value to insert */
any_type val2;   /* Second value to insert */
```

The format function operates in the same fashion as the printf function described above except that instead of printing the formatted string it returns the string as the value of the function. For example, the statements:

```
string s;
int numrec = 8;
s = format("There are %d records",numrec);
```

results in the string variable ‘s’ containing the string “There are 8 records”. Note that we did not use a “\n” operator in the pattern string because we did not want a carriage-return, line-feed sequence at the end of the resulting string.

sscanf — Scan string

Function prototype:

```
int sscanf(input,format,val1,val2,...)
string input;      /* String to be scanned */
string format;     /* Scan format pattern */
any_type val1;     /* Variable to get first value */
any_type val2;     /* Variable to get second value */
```

The sscanf() function scans the string specified as the first argument (*input*) and exacts values from the string under the direction of a format string that is specified as the second argument (*format*). The values extracted from the string are stored in the variables specified as the third and following arguments.

The value returned by the function is the count of the values scanned and assigned to variables.

For example, the statements

```
string instring = "12 34";
int i,j,count;
count = sscanf(instring,"%d %d",i,j);
```

sets the value of ‘i’ to 12, ‘j’ to 34, and ‘count’ to 2.

As with printf, the conversion operators in the format string begin with a percent sign, may contain optional field width specifications, and end with a conversion character. The following conversion characters may be specified:

- **d** — An integer value is expected to be the next thing in the input string. It is converted to an integer value and stored into the corresponding variable in the argument list. The value may consist of one or more digit and it may have an optional sign.
- **x** — Similar to integer processing except that the value is hexadecimal (for example, 1A3).
- **f** — A real (double) value is expected to be the next thing. The converted real value is stored into the corresponding variable.
- **s** — A string value is moved from the input string to the next variable. If a width is specified for the field (for example “%6s”) then the specified number of characters are moved. If no width is specified (i.e., “%s”) then the remainder of the record is moved to the next variable.
- (**space**) — When a space is encountered in the format pattern it is skipped over along with any other consecutive space characters. If the next character in the input string is a space it is skipped over along with any consecutive spaces in the input string.

(**other characters**) — If any other character is found in the format pattern a check is made to see if it matches the next character in the input string. If the characters match they are both skipped over and the scanning continues. If the next character in the input string does not match the format pattern character then the sscanf function terminates its scan.

For example, consider the following statements:

```
string instring = “5 -37”;
int i,j,count;
count = sscanf(instring,"%d %d",i,j);
```

then the variable ‘i’ receives the value 5 and ‘j’ receives –37. The returned value of the function (which is assigned to the ‘count’ variable) is 2. Since one or more spaces in the format pattern match any number of consecutive spaces in the input string, the same values would be assigned if the input string had been

```
string instring = “5     -37”;
```

Now consider the following statements:

```
string inline = “1234567”;
int i,j;
string v;
sscanf(inline,"%2d%2d%3s",i,j,v);
```

in this case there are no spaces in the format pattern but field widths are specified. The first field pattern, “%2d”, has a width of 2 so the first two characters, “12” are converted to decimal 12 and stored into the variable ‘i’. Similarly, ‘j’ gets 34, and ‘v’ gets the string “456”.

Now consider these statements:

```
string instring = “4/30/2004”;
int month,day,year;
sscanf(instring,"%d/%d/%d",month,day,year);
```

then ‘month’ gets the value 4, ‘day’ gets the value 30, and ‘year’ gets the value 2004. Note that the ‘/’ character in the format pattern matched the corresponding character in the input string and was skipped over.

The value returned by the sscanf function is the number of values which were successfully scanned and assigned to variables. If the input string does not contain enough characters to

fill all fields, the returned value of sscanf can be used to detect that. For example, consider the statements:

```
string instring = “12 23”;
int count,i,j;
count = sscanf(instring,"%d %d",i,j);
```

then ‘i’ gets the value 12, ‘j’ gets the value 23, and ‘count’ gets the value 2 because two values were successfully accrued. However, if the input string had been:

```
string instring = “12”;
```

Then ‘i’ would get the value 12, ‘j’ would get the value 0, and ‘count’ would get the value 1 because only a single input value was accrued.

fopen — Open a file

Function prototype:

```
int fopen(string filename, string mode)
```

The fopen function opens a file. The ‘filename’ argument is the file specification for the file being opened.

The value returned by the fopen function is a file “handle” number that is used by subsequent I/O functions such as fprintf, and fclose to identify the file. If the fopen function is unsuccessful, a handle number of 0 is returned. You can have many files open at once. The handle number is used to direct the I/O operations to the right file.

Text and Binary Mode Files

The ‘mode’ argument is a string containing one or more of the following characters:

- “**r**” — Open an existing file for read access. The file must exist when the fopen is executed. If ‘w’ and ‘r’ are *both* specified (“rw”), then an existing file is opened for reading and writing.
- “**w**” — Create a new file. If a file with the same name already exists it is replaced by the new file. However, if ‘w’ and ‘r’ are *both* specified (“rw”), then an existing file is opened for reading and writing.
- “**a**” — Open a file in append mode. Data written to the file will be appended to the end of it. If the specified file does not exist, a new file is created.
- “**t**” — Open file in text mode. Records consist of ASCII characters and are terminated by carriage-return, line-feed.

If you wish to open a file for update (i.e., the file must exist and you will be reading and writing it) specify “rw” as the access flags. For compatability with C, you can also specify “r+”.

File I/O example

The following example opens a file named TEST.DAT and prints each text line in the file:

```
void main()
{
    int handle,status;
    string inbuf;

/*
 *  Try to open the file.
 */
    handle = fopen("test.dat","rt");
    if (handle == 0) {
        print("Unable to open test.dat");
        stop;
    }
/*
 *  Read and print all records in the file.
 */
    loop {
        status = fread(handle,inbuf);
        if (status != 0) break;
        print(inbuf);
    }
/*
 *  Close the file and exit.
 */
    fclose(handle);
    stop;
}
```

fclose — Close a file

Function prototype:

`void fclose(int` *`handle`*`)`

The fclose function closes a file that was previously opened using the fopen function. The ‘handle’ argument is the integer file handle number returned by the fopen function.

fprint — Write line to file

Function prototype:

```
int fprint(handle,val1,val2,...)
int handle;        /* File handle number */
any_type val1;     /* First value to print */
any_type val2;     /* Second value to print */
```

The fprint function combines a series of values into a single text line and writes it to a file. It operates in the same fashion as the print function described above except that the text is written to a file rather than being written to the program log.

The ‘handle’ argument is a file handle number as returned by the fopen function. A variable number of value arguments may be specified and they may be of type string, integer, or real. Integer and real values are converted to strings before they are written to the file. If multiple value arguments are specified they are concatenated with a single blank between them to form the line. The line is terminated with a carriage-return line-feed sequence when it is written to the file.

The returned value of the function is 0 if the function is successful. An error code is returned if an error occurs during the function execution.

fprintf — Write formatted line to file

Function prototype:

```
int fprintf(handle,format,val1,val2,...)
int handle;      /* File handle number */
string format;   /* Format pattern string */
any_type val1;   /* First value to insert */
any_type val2;   /* Second value to insert */
```

The fprintf function operates in the same fashion as the printf function except that the formatted print line is written to a file rather than the program log.

The ‘handle’ argument is a file handle number returned by the fopen function Remember to specify “\n” in your format string to terminate each line.

fread — Read a record from a file

Function prototype:

```
int fread(handle,val1,val2,val3,...)
int handle;       /* File handle number */
any_type val1;    /* Variable to get first value */
any_type val2;    /* Variable to get second value */
any_type val3;    /* Variable to get third value */
```

The fread function reads a record from a file, divides the record into a series of values and stores the values into a set of variables that you specify as arguments.

When reading from a text mode file, a single record is read up to the next carriage-return, line-feed sequence. The values of the variables are separated by spaces and/or commas. If you specify a string type variable it must the the last argument. A string variable receives all characters in the line starting after any earlier numeric values.

The returned value of the function is 0 if it is successful. An error status code is returned if an error is detected.

The ‘handle’ argument is a file handle number returned by the fopen function.

The following example opens a file, reads through it, and prints each line of the file.

```
void main()
{
    int handle,status;
    string inbuf;

/*
 *  Try to open the file.
 */
    handle = fopen("test.dat","r");
    if (handle == 0) {
        print("Unable to open test.dat");
        stop;
    }
/*
 *  Read and print all records in the file.
 */
    loop {
        status = fread(handle,inbuf);
        if (status != 0) break;
        print(inbuf);
    }
```

```
/*
 *  Close the file and exit.
 */
    fclose(handle);
    stop;
}
```

fscanf — Formatted read from file

Function prototype:

```
int fscanf(handle,format,val1,val2,...)
int handle;         /* File handle number */
string format;      /* Scan format pattern */
any_type val1;      /* Variable to get first value */
any_type val2;      /* Variable to get second value */
```

The fscanf function performs the same function as the sscanf function described above except that the text line that it scans is read from a file rather than being specified as a string variable.

The ‘handle’ argument is a file handle number returned by the fopen function.

The value returned by the function is the count of the values scanned and assigned to variables.

lseek — Seek to offset in file

Function prototype:

```
int lseek(handle,offset[,origin]);
int handle;        /* File handle number */
int offset;        /* Byte position to move to */
int origin;        /* (optional) Origin of seek */
```

Normally, file read and write operations progress through a file in sequential order. However, there are some cases where it is desirable to position to a specific location in a file different than the next sequential location.

The lseek function moves the “file pointer” to a specified location so that the next read or write operation begins at that location.

The ‘handle’ argument is the file handle number as returned by the fopen function.

The ‘offset’ argument is the byte position to move to. This offset can be relative to the beginning of the file, the end of the file, or the current location.

The ‘origin’ argument determines how the ‘offset’ argument is interpreted. ‘origin’ may have one of three values:

0 — The offset is relative to the start of the file.

1 — The offset is relative to the current position. Specify a negative value for ‘offset’ to move to a position closer to the start of the file or a positive value to move further towards the end of the file.

2 — The offset is relative to the end of the file. An offset value of 0 would cause the file pointer to be positioned to the end of the file. Specify a negative value for ‘offset’ to move to a location in front of the end of the file.

The ‘origin’ argument is optional. If you omit it, it defaults to 0 (offset is relative to the start of the file).

For example, the following function call would position the file pointer to the front of the file so that the next fread function would read the first record in the file:

```
lseek(handle,0,0);
```

The value returned by the lseek function is the byte position of the file pointer (relative to the base of the file) *after* the lseek function completes its positioning. To determine the current location of the file pointer without moving it, specify 0 for ‘offset’ and 1 for ‘origin’ (i.e., position to an offset 0 bytes from the current location).

Error Status Functions

lasterror — Get last function error code

Function prototype:

```
int lasterror(void)
```

The lasterror function returns the numeric error code stored by the previous function. If there is no stored error code, a value of 0 is returned.

errormsg — Convert error code to message

Function prototype:

```
string errormsg([int code])
```

The errormsg function converts an error status code as produced by a library function into a text string that describes the error. The ‘code’ argument is the error status code to convert. If the ‘code’ argument is omitted, the last pending error code is converted to a message.

Preprocessing Directives and Macros

Introduction

DTL is a “compiled” programming language. This means that the DTL compiler reads the DTL source program that you create and generates an “object” file that can be executed.

There are several stages that the compiler goes through while compiling each statement of your program. The first stage is called the “preprocessor”. The preprocessor is responsible for reading your program from its disk file and performing certain preliminary operations on the program before passing it to the actual compiler.

The preprocessor is somewhat like an automatic editor that uses “directives” in your source program to cause it to make changes to your program. These editing changes do *not* change your source program file but occur between the time that the program is read from disk and passed to the compiler.

Note: Although the preprocessor is a very powerful and useful part of the DTL language, it is *not* necessary to use any of the preprocessor directives described in this chapter to write DTL programs. If you are just learning to program, or are just learning the DTL language, you may want to skip this chapter now and read it later once you master the essentials of writing DTL programs.

The following is a summary of the operations performed by the preprocessor:

1. Strips out comments.
2. Looks for “#include” directives which cause additional source files to be read and inserted at the point where the directive occurs.
3. Expands “macro” directives which cause text defined by earlier directives to be inserted in the source program. That is, the macro is replaced by some previously defined text.
4. Looks for “#if” directives that can cause portions of your program to be conditionally skipped over.

These operations are performed on a character-by-character basis as the source program is read. The output from the preprocessor is fed into the main body of the compiler which parses the program and generates the object file.

If you have never used a language with a preprocessor, you may find yourself getting confused between preprocessor directives and DTL language statements. Relax, you can

begin by writing DTL programs that use no preprocessor directives and then gradually advance to using simple directives and then more complex ones. All preprocessor directives begin with a '#' sign so you can always distinguish a directive from other parts of the language.

The pre-processor is told what to do through "directives" which are sometimes called "macros". Directives are commands that are specified with a '#' followed by the directive name. However, there are some circumstances where you want to use a literal '#' character in your program that is unrelated to preprocessor directives. To do this, specify the two character sequence "##" where you want a literal '#' character to be inserted. For example, the statement

```
printf("The ## of items is %d\n",count);
```

prints the string "The # of times is..." because the "##" sequence was converted to a single '#' character. If a '#' is not followed by another '#' or a letter or a digit it is interpreted as just a single '#'.

Other than DTL, there are only a few "higher level" languages that provide preprocessor directives. The best known of these is C. If you are a C programmer you will be able to quickly learn how to use the DTL preprocessor directives. However, be careful; the DTL preprocessor is somewhat different than the C preprocessor. It is much more character oriented rather than line oriented. This means that many constructs can operate within the same line, rather than across lines. Substitution can occur inside quoted strings, which is different than C. The #macro directive is more powerful than the 'C' #define form with arguments.

Examples of substitution rules

Before getting into the details of preprocessor directives, let's look at a couple of simple examples. The "#define" directive defines a name and an associated string. The form of the directive is

`#define` *`name string`*

where "name" is the name being defined and "string" is a string of characters that may consist of multiple words and spaces. For example, consider the following directive:

```
#define MAXPEOPLE 100
```

This defines the name “MAXPEOPLE” and associates it with the string “100”. At any point in your program following this directive you can cause the string “100” to be inserted in the text of the program by using “#MAXPEOPLE”. When the preprocessor sees ‘#’ followed by a name, it replaces the “*#name*” characters with the string that was associated with the name in the #define directive. For example, consider the following portion of a program:

```
#define MAXPEOPLE 100
void main()
{
    int age[#MAXPEOPLE];
    string name[#MAXPEOPLE];
    << remainder of program >>
}
```

When the preprocessor sees “#MAXPEOPLE” it replaces it with the string “100” before passing it on to the compiler. Thus, the program that the compiler sees is:

```
void main()
{
    int age[100];
    string name[100];
    << remainder of program >>
}
```

By writing your program in this way, you can change the string “100” in the #define directive and have all occurrences of #MAXPEOPLE in your program change value. Note that because preprocessor directives are processed before the compiler sees the program, you can use substitution operations at places where variables could not be used, such as in array size declarations.

In addition to words that you define using the #define directive, there are some built-in preprocessor directives that cause strings to be substituted for the directives. For example, the “#time” directive causes the preprocessor to replace “#time” with a string corresponding to the current time. Similarly, “#date” will be replaced by the current date.

```
main()
{
    string now = "Program was compiled at #time on #date.";
    printf("%s\n",now);
}
```

Assuming that the processor processed these directives on 15:15 May 23, 2005, the preprocessor would replace #time with the string 15:15:00 and #date with 05/23/2005. The value of the string passed to the compiler would be “This program was compiled at 15:15:00 on 05/23/1994.”.

The preprocessor will substitute inside or outside of quoted strings and the output of the preprocessor is not necessarily valid DTL. For example, the following use of #date and #time would produce a program which would not compile.

```
main()
{
    printf("Todays date and time are %s %s",#date,#time);
}
```

The preprocessor will expand this into

```
printf("Todays date and time are %s %s",05/23/1994,15:15:00);
```

These arguments to printf are not valid.

Including other files, the #include directive

The most common use of the preprocessor is to include one source file in another. In any significantly sized project it is very useful to separate commonly used definitions into separate source files. These commonly used definitions are then “included” into source files which need them.

The #include directive will substitute the contents of one file into another. The form of this directive is

`#include "`*`filename`*`"`

Where “filename” is the name of the file to be inserted into your source program. The file name may include device, directory, and extension portions.

For example, assume we have the two source files X1.DTL and X2.DTL with the following contents:

```
/*------------------------
 * This is the file X1.DTL
 */
string names[6] = {
    "Shelley",
    "Susan",
    "Ann",
    "Patty",
    "Helen",
    "Elaine"};
```

```
/*------------------------
 * This is the file X2.DTL
 */
#include "x1.dtl"
main()
{
    int    i;
    for (i=0; i < arraysize1(names); i++) {
        printf("Name %d is %s\n",i,names[i]);
    }
}
```

The x1.dtl file does not contain a complete program. Rather, it contains the definition of a string array initialized to six values. The x2.dtl file is a program source file that can be compiled. When the preprocessor encounters the #include directive in x2.dtl it inserts the contents of the x1.dtl file into the t2.dtl program. The resulting program which is passed to the compiler is as follows:

```
/*------------------------
 * This is the file X2.DTL
 */
/*------------------------
 * This is the file X1.DTL
 */
string names[6] = {
    "Shelley",
    "Susan",
    "Ann",
    "Patty",
    "Helen",
    "Elaine"};
main()
{
    int    i;
    for (i=0; i < arraysize1(names); i++) {
        printf("Name %d is %s\n",i,names[i]);
    }
}
```

There can be many #include directives in your program and you can use #include directives within files that are being included (i.e., you can nest #include directives).

Simple name substitution, the #define directive

The second most common use of the preprocessor is to assign a symbolic name to a constant which is used in the program.

The form of the #define is:

`#define` *`name value`*

where “name” is the symbolic name and “value” is everything else on the line up to the end of the line. Thus all of the following are valid.

```
#define pi 3.14
#define note Remember to ask dan about that new terminal type.
#define expression ((x >= 17 ? 123 : 456) || qwe() && qwe2())
#define backslash \
#define success    0        // Successful return code
```

Note that the values of the #define’s do not have to be valid DTL statements or names —

they are simply strings that will be substituted into the source text of your program. Also note that the value of the #define “backslash” is the character ‘\’. In DTL, ‘\’ at the end of a line does not mean to append the next line as it does in the C preprocessor.

In the case of the “#define success” note that there is a valid comment after the define. This comment IS NOT part of the value of “success”. Comments are stripped before #define processing occurs.

The symbolic name is substituted by specifying #*name* in the program, where “*name*” is a previously defined name. For example,

```
#define pi 3.14
    double radius,diameter;
    diameter = 10.;
    diameter = #pi * radius ** 2;
    printf("The value of pi is #pi\n");
```

Note that substitution can occur inside or outside of a quoted string. As mentioned before, two #’s (##) will collapse to a single ‘#’.

In C, the #define directive is also used to define macros with arguments. In DTL this functionality is maintained with the more flexible #macro directive

.

An advanced example

The next example demonstrates some of the more subtle aspects of name substitution. While rarely useful, the example demonstrates the order that the preprocessor performs its tasks.

1. You can have, as part of the value of one #define, the value of another define. In this example the value of ‘msg’ is part of the value of ‘errmsg’.
2. It was not necessary for ‘msg’ to be defined in order for ‘errmsg’ to want its value. While ‘errmsg’ is defined on line 1, ‘msg’ is defined on line 2. The secret to this is that the value of ‘errmsg’ is stored internally as containing the string ‘#msg’, not the expanded value.
3. When ‘errmsg’ gets expanded the first time at line 6 it gets the current value of ‘msg’ which was defined on line 2.
4. Using the #undef ‘msg’ was redefined as a different value on line 8. The ‘errmsg’ expansion on line 9 gets this second, longer value.

```
1. #define errmsg printf(#msg);
2. #define msg "This is a message"
3.
4. main()
5. {
6.     #errmsg
7. #undef msg;
8. #define msg "This is a longer message"
9.     #errmsg
10.}
```

Conditional compilation

Another use of the preprocessor is to conditionally compile sections of code. This provides an ability to block out lines of text in a source file which are not to be compiled. Frequently during development you may find that a certain portion of the program is not quite ready for compilation or needs to be fleshed out later. Conditional compilation allows you to skip over parts of a program, making them invisible to the compiler. The lines being blocked out do not even have to be valid DTL statements.

Comments, of course, can be used to block out lines in a file but if the the lines to be blocked out themselves contain comments it is more difficult.

In DTL, the #if, #ifdef, and #ifndef directives are used to conditionally compile sections of a program. The section of code to be blocked out is signalled with a #if and terminated with a #endif. There is always a pairing of these two directives. The #if, #ifdef, and #ifndef directives accept arguments which are evaluated to 1 or 0. If the result is 1 the code is passed through for compilation. If it is 0 the code is not passed through for compilation (it is discarded as if it were a comment).

There can be #if/#endif pairs nested within other #if/#endif pairs.

Within the #if/#endif there can be a #else directive. The program following the #else will be expanded if the #if condition is false. This is analogous to the behaviour of the if/else construct in DTL, although the #if processing only occurs during compilation and only affects the source lines of the program.

There is one other difference between the ‘C’ and DTL implementations of the #if suite. In ‘C’ these are strictly line oriented. The #if must be on a line by itself and it delineates a set of lines terminated by #endif. In DTL the directives are character oriented; they can block out a set of characters within a line. The examples will illustrate.

The #if directive

The form of he #if directive is

`#if (value)`

The #if directive accepts the value 1 or 0 to indicate whether the section of code is to be compiled or not. Unlike ‘C’, the value specified to #if is enclosed in parenthesis. In addition to specifying the character ‘1’ or ‘0’ you can use preprocessing lexical directives whose value is ‘1’ or ‘0’. One example is the #length directive. The value of #length(a) is the character ‘1’. Thus, #if (#length(a)) is equivalent to #if (1).

Here is a sample of the use of #if. This will demonstrate the use of the true and false conditions, as well as the use of #else.

```
#if (1)
    printf("The condition is true\n");
#endif

#if (0)
    printf("This statement will be omitted.\n");
#endif

#if (1)
    printf("The condition is true.\n");
#else
    printf("This statement will be omitted.\n");
#endif

#if (0)
    printf("This statement will be omitted.\n");
#else
    printf("The condition is true.\n");
#endif
```

Here is an example of the use of #if within a line.

```
#if (1) printf("True\n"); #else printf("False\n"); #endif
```

Here is a more complex example. Note that substitution is occurring within a line and within a quoted string.

```
printf("#if (1) True #else False #endif condition\n");
```

#ifdef and #ifndef

The #ifdef directive is another conditional compilation directive. Its form is

`#ifdef (`*`name`*`)`

where “name” is a name that may have been previously defined using a #define directive. The #ifdef directive compiles the characters that follow if the specified name has been defined. If the name has not been defined then #ifdef begins skipping over characters until the next #endif or #else directive is found.

The #ifndef directive is the reverse of #ifdef, it processes the following characters if the specified name is *not* defined. The form of this directive is

`#ifndef (`*`name`*`)`

Here is an example showing #ifdef and #else.

```
#define version_420 1

#ifdef (version_420)
    count = read(size,action_type);
#else
    count = read(size,action_type,sequence_indicator);
#endif
```

Here is another, more common use.

```
#define debug 1

#ifdef (debug)
    printf("<DEBUG> security level set to %d\n",seclevel);
#endif
```

Again note that in these examples the use of #defines and #if blocks controls the way that the program *compiles*, not the way it *executes*. Preprocessing directives affect the source program before it is compiled.

Macro definition and use

Users familiar with C will remember that the #define directive can be used to define macros with arguments. The syntactic rules which make the distinction between a macro and regular define are rather strict. In addition, multiple line macros must have their lines terminated with '\' to indicate that the next line contains more data.

The "#macro" DTL directive overcomes these limitations and is the mechanism for defining multi-line macros and macros with arguments.

The basic form of the #macro is:

```
#macro name(arg) value
```

The 'name' is the name of the macro. The '(arg)' indicates arguments to the macro. The 'value' is the value to be substituted for the macro.

The value part of the macro can be within one line or it can extend across lines. By default, the macro value is contained in a line

Macro arguments; definition and use

The arguments to a DTL macro are different than the arguments to a 'C' macro. In DTL the number of arguments to a macro are specified, not the names of the arguments as in 'C'. There can be from 0 up to 9 arguments to a macro. When the argument count is not specified it means that a variable number of arguments can be specified. If an argument count is specified the count is verified when the macro is expanded.

Within the body of the macro the arguments are referred to with the directives #0, #1, #2, etc. up to #9. The directive #0 substitutes as the name of the macro and #1 through #9 are arguments 1 through 9. The directive #argcnt is the number of arguments specified to the macro. Here are some examples which demonstrate the one line macro definition.

```
#macro noargs(0) This macro, named #0,  has no arguments.
#macro someargs(3) Macro, named #0, has arguments #1 #2 #3
#macro varargs() This macro, named #0, had #argcnt args
main()
{
    printf("#noargs()\n");
    printf("#someargs(1,2,3)\n");
    printf("#varargs(1,2,3,4)\n");
}
```

Note that even though the ‘noargs’ macro has no arguments it still requires the parenthesis after the name when it was referenced. The reason for this is consistency. Names defined with #define’s *never* require the parenthesis and names defined with #macro *always* require them. This eliminates the possibility of syntactic ambiguity.

Multiple line macros

In order to define multiple line macros the directives “#{” and “#}” are used. These have a similar use to the ‘{’ and ‘}’ characters in the DTL programming language; they group together a set of things. In the case of macro definitions the #{ and #} can be used to span lines in a macro definition. The following sample will demonstrate this.

```
#macro multiline(3) #{
/*
 * Handle this test case
 */
    switch (testcase) {
    case #1:
        test_1(#1,#3);            // Normal case
        break;
    case #2:                      // Special case
        break;
    case #3:
        printf("?Unknown state #3\n");
        break;
    }
#}
```

Note that the definition can contain comments; these are stripped and are not part of the definition. Note that the body of the macro does not require ‘\’ at the end of the line, as it would have in ‘C’. The value terminates at the “#}”. Although the “#{” and “#}” were at the top and bottom of the definition for readability, it is not necessary for them to be on separate lines. The following is valid, if not as readable:

```
#macro testit(2) #{ if (#1 == 16 || #2 == 19) {
printf("I am unable to process this at this time); } #}
```

Lexical directives

DTL lexical directives return values that are strings. The “value” of the #date directive is, for instance, the string consisting of the date.

In that light there are certain directives which manipulate strings and return strings as results. The directives can use other directives as arguments. The internal directives are evaluated first, then the outer. In all cases, the lexical functions accept arguments within parenthesis and evaluate those arguments character by character. The directive #length, for instance, returns the length of the string in parenthesis. The value of #length( ) is 1 since there is a single blank. The value of #length(ab) is 2, the ‘a’ and the ‘b’. Spaces are significant (counted) in #length argument strings.

#cmpeq and #cmpne

These two directives can be used to compare two strings. The directives return the value 1 or 0, depending on whether the strings match or not. The form of these directives is:

#cmpeq(*string1*,*string2*)
#cmpne(*string1*,*string2*)

Where ‘string1’ and ‘string2’ are the items to be compared and #cmpeq returns 1 if the strings are equal and #cmpne return 1 if the strings are not equal.

Here are some examples of #cmpeq. #cmpne returns opposite results. Note the sensitivity to spaces in the string comparison.

```
#cmpeq(abc,def)        --> 0
#cmpeq(abc,abc)        --> 1
#cmpeq(abc ,abc)       --> 0
#cmpeq(abc , abc)      --> 0

#define backslash \
#cmpeq(\,#backslash)   --> 1

#cmpeq(#length(abc),3) --> 1
```

#quote

The #quote directive will replace its string argument with the argument surrounded by quote signs. This is very useful in a macro for creating a quoted string. Note that if the argument is already a quoted string #quote will place another pair around it.

The form of #quote is:

`#quote(`*`argument`*`)`

Some examples and their results:

```
#quote(1)                                  --> "1"
#quote(1,2,3)                              --> "1,2,3"
#quote("string")                           --> ""string""
#quote(#concat(a,b) #length(#date))        --> "ab 10"
```

#length

The #length directive returns the length of the string which is its argument. Spaces are significant (counted) in the #length argument string. The form is

`#length(`*`arg`*`)`

The length of the argument string is evaluated and the directive is replaced with a string corresponding to the numeric length. Some samples and what they generate shown below:

```
#length(123)                --> 3
#length("ABC")              --> 5
#length(#date)              --> 10
#length(#length(#date))     --> 2
```

Think about that last one. The length of #date is 10 (mm/dd/yyyy). The length of 10 (which is itself a string containing ‘1’ and ‘0’) is 2. The argument to the macro is evaluated before the macro is expanded.

Here is an example of a macro definition that uses the #length directive:

```
#macro arglength(1) #{
    printf("The length of arg1 is #length(#1)\n");
#}
```

In this example the macro will generate a printf which contains the length of the argument to the macro.

Since lexical directives can use other directives as arguments, the value of #length(#include “x.c”) is the length of the file ‘x.c’.

#concat

The #concat directive will concatenate a series of strings into a single string. This can be very useful in macros for creating new variable or precedure names

The form of #concat is

```
#concat (s1,s2,...)
```

#concat accepts a variable number of arguments. Here are some samples. Note that an argument to a lexical directive can be a sinqle quote character. Quoted strings are not maintained as separate items as they are in ‘C’.

```
#concat()                                  -->
#concat(1,b)                               --> 1b
#concat(1, ,b, ,c)                         --> 1 b c
#concat(1,b,"xyz")                         --> 1b"xyz"
#concat(1,b,#length(this is a string)) --> 1b16
#concat(",this is a string,")              --> "this is a string"
```

Error handling

There are directives which can be used in error processing. They are useful inside macros to validate argument values or validate symbol values. All operate in a similar fashion but produce different messages. They accept a string argument which is echoed to the terminal.

- #info (*msg*) produces a message defined as informational.
- #warn (*msg*) produces a message defined as a warning.
- #error (*msg*) produces a message defined as a error.
- #fatal (*msg*) produces a message defined as fatal. A fatal message will result in an abort of the compile.

An example use might be:

```
#ifndef (version_4)
    #fatal (Symbol version_4 must be defined for this compile)
#endif
```

Or perhaps

```
#info (Compiling #file. Remember to record source changes)
```

Miscellaneous directives

There are several other directives, performing various functions.

- #undef is used to ‘undefine’ a name defined via #define or #macro.
- #date has the value of the current date in the form mm/dd/yyyy.
- #time has the current time (the time the compile is performed) in the form hh:mm:ss where the hours are military time.
- #file is the name of the current source file.
- #line is the current line number of the current source file.

Advanced macro design

Using lexical directives as well as the error handling it is possible to design complex macros which perform string manipulation and comparison and conditional expansion. While some of these examples offer little practical use, the knowledge of such abilities can strengthen knowledge of macro use.

Use of comparison and conditionals

The arguments to a macro, remember, are #0 through #9. Using the #cmpeq and #cmpne directives and the #if extensive argument verification can be performed.

```
#macro build_list(3) #{
    #if (#cmpne(#1,arg1))
        #warn(Expected arg1 as first argument, got #1)
    #endif
    #if (#cmpne(#2,arg2))
        #warn(Expected arg2 as second argument, got #2)
    #endif
    #if (#cmpeq(7,#length(#3)))
        #info(Argument 3 to build_list is of length 7)
    #endif
#}
```

Use of other macros within macros

Macros can invoke other macros. Macros can even *define* other macros. Common macros can be used between other macros. Here are some samples.

```
/*
 * Define a set of defines. The defines will take
 * value when the macro is invoked.
 */
#macro mass_define(0) #{
#define md_1    1
#define md_2    2
#define md_3    3
#define md_4    4
#}
```

Here we use one macro to invoke a set of other macros.

```
/*
 * Invoke a set of initialization macros
 */
#macro macro_init(4) #{
    #init1(#1)
    #init2(#2)
    #init3(#3)
    #init4(#4)
#}
```

Use of lexical functions to manipulate arguments

The #concat and #quote can be very useful for constructing strings from macro arguments as shown in the following example.

```
#macro debug_check() #{
/*
 * Handle one and two argument cases separately
 */
    #if (#cmpeq(#argcnt,1))
        printf(#quote(#concat(DEBUG: ,#1,\n)));
    #endif

    #if (#cmpeq(#argcnt,2))
        printf(#quote(#concat(TEST: ,#1,\n)));
        printf(#quote(#concat(TEST: ,#2,\n)));
    #endif
#}
```

Index