Chapter 7

Declaring Program Variables

Chapter Overview

This chapter discusses the use of program variables in a CL program. You learn the rules about naming variables, the types of variables CL supports, and how these variables are declared in a CL program.

What Is a Program Variable?

A **program variable** is a named field used to store a value. For example, a variable named &filename might contain the value CUSTFILE, or a variable named &custnbr might be used to store the value 53124. Unlike a **constant**, which has a value that never changes, the value stored in a variable can change every time a CL program is run. You also can change a variable's value many times during the execution of a CL program. A typical use of a CL program variable might be to act as a counter within a program to control how many times a loop is processed.

You can also use variables within CL commands as substitutes for nearly all command parameters. For example, instead of specifying the following command in a CL program

```
DSPTAP  DEV(TAP01)
```

you could replace the fixed value (TAP01) in the DEV parameter with a variable named &tapedevice:

```
DSPTAP  DEV(&tapedevice)
```

The value of variable &tapedevice would be determined when the program is executed. In effect, the DEV parameter of command DSPTAP could contain a different value each time the program is run, adding versatility to the command's use in your program.

Program variables exist only within the program in which they are used; when the program ends, the values stored within the variables are no longer accessible. You can use variables to add flexibility to your CL programs and to pass information between programs.

How to Declare a Variable

For a variable to be used in a CL program, the variable must be defined. You define a variable name and its characteristics, such as data type and length, with the DCL or DCLF command. All DCL and DCLF commands must appear in the declarations section of a CL source member.

The DCL Command

The **DCL** (Declare CL Variable) command explicitly defines an individual variable in your program. For example, the command

```
DCL  VAR(&dayofweek)  +
     TYPE(*CHAR)      +
     LEN(9)
```

defines a variable named &dayofweek, which a program could use to store the day of the week. The variable is defined to hold character data (*CHAR) and is nine characters long. This declaration lets you store a value such as "Wednesday" in the &dayofweek variable.

Naming Variables

The first parameter on the DCL command, VAR, is required. The **VAR (Variable name) parameter** assigns a name to the variable. Every variable name begins with an ampersand (&), followed by one to 10 characters that uniquely identify the variable. Permitted characters following the & are A–Z, a–z, @, #, or $. The remaining characters can also include the numbers 0–9 and the underscore (_) character. Variable names cannot contain embedded blanks or other special characters, such as !, %, or *.

Specifying the Data Type

The **TYPE parameter** on the DCL command also is required. This parameter specifies the type of data the variable can contain. CL supports six data types when declaring variables:

- *CHAR for variables containing character data
- *LGL for variables containing logical data
- *DEC for variables containing numeric data in packed decimal format
- *INT for variables containing numeric data in signed integer format
- *UINT for variables containing numeric data in unsigned integer format
- *PTR for pointer variables containing a memory address

Variables of **character type** (*CHAR) can be used to hold any character or string of characters. The system stores the characters as consecutive bytes, one byte per character.

Logical type (*LGL) variables, sometimes called *flags*, *switches*, or *indicators*, are a specialized type of character data used to store a Boolean data value. Few programming languages use Boolean data values, and this type may be new to you. Boolean logical variables are stored as single-byte fields that can contain only a '1' (on) or a '0' (off), to indicate either a true or false condition.

Decimal type (*DEC) variables are typically used to hold numeric data. The numeric data is stored in packed decimal format within the program's storage. CL does not support zoned decimal variables.

Integer (*INT) and **unsigned integer** (*UINT) variables also hold numeric data. The numeric value is stored in a compact binary format that is common among many computer platforms. Integers are whole numbers (or zero), with no decimal digits. An *INT variable includes a sign, while an *UINT variable is always positive (or zero).

Pointer type (*PTR) variables hold the memory address of the data being referred to and not the actual data.

Specifying the Variable Size

The DCL command's optional **LEN (Length) parameter** specifies the variable's length. The length allowed for a variable depends on its data type. Although the length parameter is not required, it is nearly always used for character and numeric variables. If the parameter is not specified, a default length is assigned to the variable. The following table summarizes the minimum, maximum, and default lengths for each data type.

Data type	Minimum/maximum length	Default length
*CHAR	1–32,767 bytes	32 bytes
*LGL	1 byte	1 byte
*DEC	1–15 digits (0–9 decimals)	15 digits (5 decimals)
*INT, *UINT	2 or 4 bytes	4 bytes
*PTR	Length cannot be specified; pointers have a length of 16 bytes.	

To specify the length for a *CHAR, enter the maximum number of characters you want the variable to hold. For example:

```
DCL  VAR(&mychar)  +
     TYPE(*CHAR)   +
     LEN(4Ø)
```

Because *LGL type variables are always one character long, you usually do not specify their length. Instead of

```
DCL  VAR(&myflag)  +
     TYPE(*LGL)   +
     LEN(1)
```

you would omit the LEN parameter from the specification:

```
DCL  VAR(&myflag)  +
     TYPE(*LGL)
```

For *DEC variables, enter the maximum number of *digits* the variable can hold. For example:

```
DCL  VAR(&mynumber)  +
     TYPE(*DEC)      +
     LEN(5)
```

You can specify an assumed decimal point for *DEC variables. To indicate where the decimal point should be placed, you use a different form of entry for the LEN parameter, specifying the length by using a pair of integers separated by a blank. The first integer defines the length of the variable; the second integer indicates how many digits within the variable fall to the right of the decimal point. For example, the following command defines a number 15 digits long with five digits to the right of the decimal point:

```
DCL  VAR(&mynumber)  +
     TYPE(*DEC)      +
     LEN(15 5)
```

The variable defined here could hold a number up to 9,999,999,999.99999. Up to nine digits may appear to the right of the decimal point. If you specify a digit length for a *DEC variable but not a number of decimal positions, the variable will default to zero decimal positions.

Integer (*INT) and unsigned integer (*UINT) variables must be either two or four bytes long and do not have any decimal places. For integers, the length should reflect the number of *bytes* allocated for the variable, not the number of digits:

```
DCL  VAR(&mynumber)  +
     TYPE(*INT)      +
     LEN(2)
```

Coding in Style

CL doesn't care whether you name variables using uppercase or lowercase characters. It will treat **&DAYOFWEEK** and **&dayofweek** as the same variable. To make your source code more readable, however, type your variable names in lower case.

In addition, the DCL command is more readable if you omit the following common keywords, which represent positional parameters:

- VAR
- TYPE
- LEN

For example, instead of using keyword notation, such as

```
DCL  VAR(&custnbr)  +
     TYPE(*DEC)     +
     LEN(5 0)
```

use the following positional notation:

```
DCL  &custnbr  *DEC  (5 0)
```

For *DEC variables, because the LEN parameter is a list-type parameter (i.e., the parameter includes more than one value), you must enclose the two integers in parentheses regardless of whether you use keyword or positional notation.

Initializing the Value of a Variable

You can assign an initial value to a variable by using the optional **VALUE (Initial value) parameter** on the DCL command. For example, if you specify the following command, you would be declaring the &dayofweek variable and assigning it an initial value of "Wednesday".

```
DCL  &dayofweek  *CHAR  9  VALUE('Wednesday')
```

This example uses positional notation for the VAR, TYPE, and LEN parameters. VALUE is the DCL command's fourth and final positional parameter, but most programmers prefer to use keyword notation for it because the third parameter (LEN) is optional.

If you don't specify the VALUE parameter, variables are automatically initialized to their default values. For *CHAR variables, the default initial value is blanks. For *LGL variables, the default initial value is '0'. For *DEC, *INT, and *UINT variables, the default initial value is zero. You cannot specify the VALUE parameter for *PTR variables.

When you supply an initial value for a variable, you must specify a value of the same type as the TYPE parameter. For example, to initialize a *CHAR variable, you must specify a character string as the VALUE. A *DEC variable, however, requires a numeric VALUE.

Assigning Values to *CHAR Variables

Assigning an initial value to a *CHAR variable is fairly straightforward. The value can consist of a string of any combination of characters, with few restrictions. The following are valid declarations:

```
DCL  &myname  *CHAR  15         +
     VALUE('This is my name')

DCL  &myname  *CHAR  15         +
      VALUE('THIS/IS=MY%NAME')

DCL  &mynumber  *CHAR  5  VALUE('12345')
```

You may have noticed in these examples that the string of characters defining a variable's value is enclosed in apostrophes ('). (Many programmers refer to the apostrophe as a *single quote*, and some call it a *tick mark*.) This format is called a **quoted string**. The character value you specify within the apostrophes can consist of any alphabetic, alphanumeric, or special characters. When an apostrophe is to appear within a quoted string, you must enter two apostrophes:

```
DCL  &end  *CHAR  16            +
     VALUE('That''s all folks')
```

Although not required, you may find that always enclosing a character value in apostrophes makes good sense. If you do not use a quoted string, you will need to remember the following restrictions, which affect the value's validity. Character values in unquoted strings

- Cannot consist solely of numbers
- Cannot contain a blank
- Cannot include CL special characters (`e.g.`, `*`, `&`, `/`, `(`, `)`, `.`, `+`, or `-`)

For example, the following command is invalid because the value in the unquoted string is considered to be a list of values (due to the presence of the blanks), not a single value:

```
DCL  &myname  *CHAR  15      +
     VALUE(This is my name)
```

You must be especially careful to include numeric values in quoted strings if they are to be the initial values for *CHAR variables. In the following examples, the unquoted numeric values are considered to be numeric data, not character data, so these examples are invalid:

```
DCL  &mynumber  *CHAR  5  VALUE(12345)
DCL  &mynumber  *CHAR  5  VALUE(-12345)
DCL  &mynumber  *CHAR  5  VALUE(+12345)
```

You also cannot specify another CL variable when assigning the initial value of a variable. And because CL reserves the ampersand character to identify variables, you cannot use the & unless you include it in a quoted string. For example, the following command is invalid:

```
DCL   &username  *CHAR  7  VALUE(&myname)
```

If you want to include the ampersand in the value for a variable, you must specify the value as a quoted string. For example:

```
DCL   &username  *CHAR  7  VALUE('&MYNAME')
```

However, be aware that this does *not* assign the value of the variable &MYNAME to the variable &username; it assigns the character string '&MYNAME' instead.

Occasionally, you may need to initialize a *CHAR variable to a value that cannot be represented by a character from your computer keyboard. In these cases, you can set the value of the variable to a **hexadecimal number**. The following command assigns a hexadecimal value to a character variable.

```
DCL  &hexnumber  *CHAR  1  VALUE(X'FC')
```

The set of hexadecimal numbers contains the values 0–9 and A–F. Hexadecimal numbers, although not often used in CL, can be employed for such tasks as manipulating screen attributes, processing lists of values, and assisting in data comparison operations. To specify hex notation, CL uses an X followed by a quoted character string consisting of pairs of hexadecimal numbers. For example, if you want to initialize a *CHAR variable to the hexadecimal value FFFF, you would enter the following DCL statement:

```
DCL  &hexff  *CHAR  2  VALUE(X'FFFF')
```

Assigning Values to *LGL Variables

The rule for specifying an initial value for a *LGL variable is easy. An important point here is that logical variables can hold only '0' (false) or '1' (true). These are character values, not numeric, so you must enclose them in apostrophes. The initial value must be character value '0' or '1', as the following examples illustrate.

```
DCL  &myflag  *LGL  VALUE('0')
DCL  &myflag  *LGL  VALUE('1')
```

The following examples break the true/false rule, so they are not valid.

```
DCL  &myflag  *LGL  VALUE('2')
DCL  &myflag  *LGL  VALUE(0)
DCL  &myflag  *LGL  VALUE(1)
```

Assigning Values to Numeric Variables

When you want to assign an initial value to a *DEC variable, you must use a numeric value: the numbers 0–9, the period (.), and the positive (+) and negative (–) signs. You cannot use a comma (,) to separate millions or thousands. However, to conform to international conventions, a comma can be used in place of a period to represent a decimal point. Only one period (or comma) may appear in a numeric value to specify the position of the decimal point. If you include a sign (+ or –) in the numeric value, it must appear as the first character of the value. The following examples are valid declarations of initial values for *DEC variables.

```
DCL  &mynumber  *DEC (5 0)  VALUE(523)
DCL  &mynumber  *DEC (5 2)  VALUE(3.14)
DCL  &mynumber  *DEC (5 2)  VALUE(-256.78)
DCL  &mynumber  *DEC (5 2)  VALUE(+256)
```

You cannot use a quoted string as the initial value for a *DEC variable, nor can you assign alphabetic characters. The following commands are invalid.

```
DCL  &mynumber  *DEC (5 0)  VALUE('523')
DCL  &mynumber  *DEC (5 0)  VALUE(ABC)
```

If you include a sign in the initial value, it must precede the numeric value. The following command is invalid:

```
DCL  &mynumber  *DEC (5 0)  VALUE(25677-)
```

You also must ensure that the decimal places in the initial value will fit into the declared length. The following examples are invalid because the values indicate a decimal alignment different from the decimal positions specified in the LEN parameter.

```
DCL  &mynumber  *DEC (5 0)  VALUE(256.13)
DCL  &mynumber  *DEC (3 3)  VALUE(1.23)
```

Integers (type *INT and *UINT) follow the same basic rules for initial values as *DEC variables, with a few additional restrictions. The integer value cannot include a decimal character, and if the variable is type *UINT, it cannot include a sign. The integer value must fall within the range allowed for the length of the integer. For example, the value of a two-byte *INT variable must fall in the range –32768 to +32767; a two-byte *UINT value must be between 0 and 65535. Four-byte integers have extended ranges. The following examples correctly assign initial values to integer variables.

```
DCL  &mynumber  *INT   2  VALUE(-256)
DCL  &mynumber  *UINT  2  VALUE(1)
```

Miscellaneous Rules for Initial Values

When you specify an initial value for a variable, you must not only ensure that you use the correct data type; you also must take into account the length of the initial value. If you specify a value on the length (LEN) parameter that is too small to hold the assigned value, the command is invalid. If the specified length is larger than that of the value, the CL compiler appends blanks or zeros to the value to match the specified length. For *CHAR variables, the initial value will be left-adjusted within the variable and padded to the right with blanks. In the following example, the variable &myname will be initialized to the value MARY~~bbbbbbbbbbbbbbbbbbbbbbbbbb~~ (where ~~b~~ indicates a blank):

```
DCL  &myname  *CHAR  30  VALUE('MARY')
```

For *DEC variables, the value will be placed in the variable according to the decimal alignment specified and, if necessary, padded to the right and left with zeros. For example, the following command initializes the variable &mynumber to the value 00016.10:

```
DCL  &mynumber  *DEC  (7 2)  VALUE(16.1)
```

You should note that if you specify the VALUE parameter of the DCL command without specifying the LEN parameter, CL sets the length of the variable according to the length of its VALUE, ignoring the default lengths shown earlier in the chapter. For example, the following command assigns a length of seven bytes to the variable &mylibrary, even though a *CHAR variable with an unassigned length defaults to 32 bytes.

```
DCL  &mylibrary  *CHAR  VALUE('PGMTEST')
```

The following command assigns a length of three bytes, with two decimal places, to the variable &pi, just as if you were to specify `LEN(3 2)`.

```
DCL  &pi  *DEC  VALUE(3.14)
```

To improve the readability and maintainability of your CL programs, you should never assign an initial value to a character or decimal variable when the length (LEN) is not specified.

You cannot assign an initial value to a variable if the variable name being declared also appears in the PARM parameter of the PGM command in the program linkage section. In this case, the value of the variable is determined at run time by a value passed to the program from another program. The following is an invalid combination of code because the program must receive the initial value of the &mylibrary parameter when the program is called.

```
PGM  PARM(&mylibrary)
DCL  &mylibrary  *CHAR  10  VALUE('MYLIBRARY')
```

Using DCL with Pointers

Pointer type (*PTR) variables are used to hold the computer storage address of a data object or function. Pointers will tell your program where a variable is stored and let you "point" a variable to a specific storage address. Pointers are not heavily used in CL programming, but a basic knowledge of how to declare and use pointers will help you when you are using application programming interfaces (APIs). The IBM i operating system supplies hundreds of APIs to access many operating system functions and data.

Pointer variables do not allow a LEN entry when you declare them. Pointers always have a length of 16 bytes and contain a computer storage address. To declare a pointer, would you simply code

```
DCL  VAR(&mypointer)  +
     TYPE(*PTR)
```

You cannot use the VALUE parameter to assign an initial value to a pointer. Instead, the DCL command supports an **ADDRESS parameter**, which you can use to point to the address of another variable. The following example initializes the variable &mypointer to the address of the variable &myname.

```
DCL  &myname     *CHAR  10  VALUE('JOHN ADAMS')
DCL  &mypointer  *PTR       ADDRESS(&myname)
```

The ADDRESS parameter also lets you enter an **offset** — a number of bytes from the beginning of the referenced variable where you want the pointer to point. In the following example, &mypointer would point to the sixth character of &myname.

```
DCL  &myname     *CHAR  10  VALUE('JOHN ADAMS')
DCL  &mypointer  *PTR       ADDRESS(&myname 5)
```

In this case, &mypointer would point to the first A in 'JOHN ADAMS' — that is, five bytes to the right of the first byte in &myname. The offset can be any value from 0 to 32766. The ADDRESS parameter is valid only with *PTR variables.

Declaring Based Variables

The concept of based variables is tied to the concept of pointers. Normally, when you declare a variable, the program will automatically allocate storage to hold the variable's value. It is possible, however, to declare a variable without allocating any storage to it. Instead, the variable's value will depend upon the value stored at an address location indicated by a pointer. The resulting **based variable** always depends upon its associated pointer to reveal the variable's value. Because your program can change the value of the pointer while it is running, it can use based variables to manipulate arrays of values or to map variables in the program to other variables.

To declare a based variable, you use the **STG (Storage)** and **BASPTR (Basing pointer variable) parameters** of the DCL command:

```
DCL  &mypointer  *PTR

DCL  &basedvar   *CHAR  10  +
     STG(*BASED)            +
     BASPTR(&mypointer)
```

This example declares a character variable (&basedvar) that is based on the &mypointer variable. The value of &basedvar will be the 10 bytes beginning at the location addressed by &mypointer. Before your program can use &basedvar, &mypointer must be initialized to a valid address using the %ADDRESS function discussed in Chapter 8. If &mypointer does not contain a valid address when you try to use &basedvar, an error will occur.

Declaring Defined Variables

A **defined variable** enables a CL program to give a name to a portion of an already declared variable or to allow two different variables to share the same storage area. Defined variables make it easy for your program to "break down" variables that may contain complex data structures. As with based variables, CL does not allocate separate storage for defined variables. Instead, the defined variable relies upon the storage allocation of another variable that is declared to your program.

To declare a defined variable, you use the STG and **DEFVAR (Defined on) parameters** of the DCL command:

```
DCL  &myobj  *CHAR  20

DCL  &myfile  *CHAR  10  +
     STG(*DEFINED)       +
     DEFVAR(&myobj)

DCL  &mylib   *CHAR  10  +
     STG(*DEFINED)       +
     DEFVAR(&myobj  11)
```

In this example, &myobj is a 20-byte character field. This variable is actually a data structure composed of two elements. The first 10 bytes are being used to name a file, while the second 10 bytes name the library in which the file exists. To refer to each of these elements separately, two defined variables are declared. The &myfile variable is 10 bytes long, and it shares the same storage as the first 10 bytes of &myobj (the DEFVAR value defaults to starting with position 1). The &mylib variable occupies the final 10 bytes of &myobj (beginning in position 11). Variable &myobj provides the storage for both &myfile and &mylib.

You can also employ a defined variable to use the same storage, but with different data types:

```
DCL  &mychar  *CHAR 4

DCL  &mybin   *INT  4  +
     STG(*DEFINED)     +
     DEFVAR(&mychar)
```

Your CL program can then use whichever variable (&mychar or &mybin in this case) best suits its needs (*CHAR or *INT) to process the same area in storage.

The next example uses a variable as a data structure, with subfields of different data types.

```
DCL  &mystruc  *CHAR 52

DCL  &mybin    *INT   2  +
     STG(*DEFINED)      +
     DEFVAR(&mystruc)

DCL  &mychar   *CHAR 50  +
     STG(*DEFINED)      +
     DEFVAR(&mystruc 3)
```

In this structure (&mystruc) the first two bytes can be processed as an *INT variable (&mybin), while the remaining 50 bytes will be treated as a *CHAR variable (&mychar). Chapter 8 discusses manipulating the values of variables, including defined variables.

More DCL Examples

Let's examine some additional examples of DCL command usage, applying the rules you have learned. An explanation of each example follows each set of DCL commands below.

```
DCL  VAR(&state)  TYPE(*CHAR)  LEN(2)  VALUE('  ')
DCL  VAR(&state)  TYPE(*CHAR)  LEN(2)
DCL  &state  *CHAR  2  VALUE('  ')
DCL  &state  *CHAR  2  VALUE(X'4040')
DCL  &state  *CHAR  2
```

The preceding DCL command examples define a variable named &state that is two characters long and initialized to blanks. (The hexadecimal representation of the blank character is '40'.)

```
DCL  &state  *CHAR  2  VALUE('CA')
DCL  &state  *CHAR  2  VALUE(CA)
DCL  &state  *CHAR  2  VALUE(ca)
DCL  &state  *CHAR  2  VALUE(X'C3C1')
```

The four examples above define the variable &state and initialize its value to 'CA'. In the third command, CL converts the lowercase 'ca' to uppercase characters because the value is not enclosed in single quotes.

```
DCL  &state  *CHAR  2  VALUE('Ca')
```

This example defines the variable &state with an initial value of 'Ca'. This time, CL does not convert the case of the value because the value is a quoted string.

```
DCL  &name  *CHAR
```

In this example, the &name variable defaults to 32 characters long, initialized to blanks.

```
DCL  &name  *CHAR  VALUE('Mr. Jones')
```

Here, the &name variable is nine characters long, initialized to 'Mr. Jones':

```
DCL  VAR(&profit)  TYPE(*DEC)  LEN(7 2)  VALUE(00000.00)
DCL  &profit *DEC (7 2) (0)
DCL  &profit *DEC (7 2)
```

Each of these examples defines &profit as a decimal variable with a total length of seven digits, two of them following the decimal point. The initial value in each example is zero. The LEN parameter is enclosed in parentheses in all the examples to group the two integers that define the length and number of decimal positions.

```
DCL  &phone  *DEC  10
```
This example declares &phone as a decimal variable 10 digits long, with no decimal places, initialized to zeroes:

```
DCL  &weight  *DEC
```
In this example, the variable &weight defaults to 15 digits long, with five digits to the right of the decimal point. The initial value is zero.

```
DCL  &pi  *DEC  VALUE(3.14)
```
This example defines the variable &pi as three digits long, with two digits to the right of the decimal point. The initial value is 3.14.

```
DCL  VAR(&FLAG)  TYPE(*LGL)  LEN(1)  VALUE('0')
DCL  &flag  *LGL  1
DCL  &flag  *LGL
```
The preceding examples define the variable &flag as a logical variable, initially off (or false).

```
DCL  &flag  *LGL  VALUE('1')
```
In this last example, the logical variable &flag is initially on (or true).

The DCLF Command

A CL program can process display files or database files. You can use a **display file** to send a screen to a workstation and to receive from the workstation information that you will use in the program. The display file would contain the screen layout, including the fields that you can display or key into. You also can use your CL program to read records from a **database file**. For database files, only input operations are allowed. You cannot use a CL program to write or update a record in a database file.

If your CL program uses either a display file or a database file, you must declare the name of the file to the program using the **DCLF** (Declare File) command. For example, the command

```
DCLF  FILE(PAYROLL/EMPMAST)
```

makes the database file EMPMAST (an employee master file) in library PAYROLL available for processing in a CL program. When you declare a display or database file in a CL program, the file must exist on your system before you can compile the program successfully.

In the example above, if the file EMPMAST contains fields named EMPNBR, NAME, and RATE, the CL compiler automatically declares the variables &EMPNBR, &NAME, and &RATE for use in your program. These variables would have the same general attributes as the fields in the file — character fields are *CHAR variables, numeric fields become *DEC variables, and the CL variables have the same lengths as the database fields.

A CL program can use up to five files; it follows, then, that as many as five DCLF commands are allowed in a CL program. In that case, each DCLF command must specify an **open file identifier**, a name to uniquely identify each declared file; if the program declares only one file, the open file identifier is optional. The following example uses the **OPNID (Open file identifier) parameter** to supply the open file identifier:

```
DCLF  EMPMAST  OPNID(EMPMAST)
```

This command makes the EMPMAST file (in the program's library list) available to the CL program with an open file identifier of EMPMAST. The open file identifier need not match the file name, but most programmers prefer to make them the same; the OPNID must follow the rules for a simple name. When you declare a file with an open file identifier, CL prefixes the names of the CL variables from that file with the open file identifier and an underscore (_) character. In this example, the fields EMPNBR, NAME, and RATE would be known as &EMPMAST_EMPNBR, &EMPMAST_NAME, and &EMPMAST_RATE in the CL program. A variable name that is prefixed in this manner is said to be a **qualified name**.

Be careful to make the distinction between processing a file in your program and using your program to perform an action using a file object. For example, you can use the **CPYF** (Copy File) command to copy a file without declaring the file in your program. If, however, your CL program performs read operations from the file, you must declare the file. Chapter 15 discusses the CL commands to process files.

DCLF Examples

The following examples will help you better understand how the DCLF command functions.

```
DCLF  FILE(INVENTORY/STOCK)
DCLF  INVENTORY/STOCK
DCLF  *LIBL/STOCK
DCLF  STOCK
DCLF  STOCK  OPNID(STOCK)
```

The examples above make the fields in file STOCK available to your program as variables. The first two examples explicitly state that the file is found in library INVENTORY; the last three examples assume that file STOCK will be found in a library in your job's library list. When you compile the program, any fields in file STOCK will be declared as variables in your CL program automatically. Because the last example specifies an OPNID, the variable names from STOCK will be qualified with "STOCK_" in the CL program.

```
DCLF  CUSTDSP
DCLF  CUSTDSP  OPNID(DISPLAY)
```

These two examples both make available to your CL program all the fields defined in display file CUSTDSP. Because no library is specified for file CUSTDSP, the compiler will search for the file in your job's library list. When the compiler finds the file, all fields in the file will be declared for use in the CL program automatically. In the latter example, the CL program will qualify the variable names with "DISPLAY_".

Tip

You may encounter rare cases when a CL program will need to refer to the same file in multiple DCLF commands. As long as each DCLF command specifies a different open file identifier, CL will allow multiple declared instances, with each instance having its own set of variables:

```
DCLF  EMPMAST  OPNID(F1)
DCLF  EMPMAST  OPNID(F2)
```

This program would refer to variables &F1_EMPNBR, &F2_EMPNBR, and so on to associate each variable with its appropriate open file identifier.

Chapter Summary

Program variables are fields within a CL program used to store values. Program variables exist only within the program in which they are used; when the program ends, the values stored in the variables are no longer accessible.

You must declare (define) all variables to the CL program before the program can use them (in other words, the program must define the variable name and its characteristics, such as data type and length). You use the DCL (Declare CL Variable) command or the DCLF (Declare File) command to declare CL variables.

The DCL command defines an individual variable; the DCLF command defines all variables in a display file or a database file. The DCL and DCLF commands must appear in the declarations section of a CL program.

Variable names always begin with an ampersand (&), followed by one to 10 characters that uniquely identify the variable (e.g., &dayofweek, &name, &number).

CL supports six data types for variables: *CHAR (character), *LGL (logical), *DEC (decimal), *INT (integer), *UINT (unsigned integer), and *PTR (pointer). If you do not specify a variable's length when you declare it, CL will assign a default length to the variable. If you don't assign an initial value to a variable, CL provides an initial value appropriate to the variable's data type.

Based variables depend on an associated pointer variable for processing. Defined variables enable CL to declare data structures or to allow two variables to share the same storage.

CL can process both display files and database files. A CL program can read the records in a database file, but it cannot update them or write new ones. A CL program can declare up to five files, using the DCLF command; it distinguishes each file by assigning an open file identifier to each one. When a file has an open file identifier, CL qualifies the file's variable names with the open file identifier.

Key Terms

*CHAR
*DEC
*INT
*LGL
*PTR
*UINT
ADDRESS parameter
based variable
BASPTR parameter
character type
constant
CPYF
database file
DCL
DCLF
decimal type
defined variable
DEFVAR parameter
display file
hexadecimal number
integer type
LEN parameter
logical type
offset
open file identifier
OPNID parameter
pointer type
program variable
qualified name
quoted string
STG parameter
TYPE parameter
unsigned integer
VALUE parameter
VAR parameter

Review Questions

1. What is a CL program variable?
2. Why are CL program variables needed?
3. Can you declare a CL program variable with the same name twice in a program?
4. How and where do you declare variables in a CL program?
5. CL provides two commands to declare variables. Name the commands, and explain their function.
6. What data types does CL support when declaring variables?
7. The OPNID parameter was added to CL at V5R3. What advantages has this parameter given the CL programmer?
8. Explain what a based variable is.
9. Why would you use a based variable?
10. What is the length of a pointer variable?
11. Summarize the rules for naming a pointer variable.
12. List the value assigned to each variable data type if the variable is not initialized.
13. What does the CL compiler do if you initialize a variable value without specifying the length of the variable?

Exercises

1. Determine which of the following variable names are valid and which are invalid. If the variable name is invalid, explain why.

    a. `&MEMBER`
    b. `&12345`
    c. `*NUMBER`
    d. `&1FIRST`
    e. `&NUMBER1`
    f. `&MYFIRSTNUMBER`
    g. `&@@@@A`
    h. `@&&MASTER`
    i. `&#@$_16`
    j. `&my_number`
    k. `&dues`
    l. `&john doe`
    m. `&$dollars`

2. Determine the initial value and length of the following variables.

   a. `DCL  &number  *DEC  VALUE(16)`
   b. `DCL  &name  *CHAR  10`
   c. `DCL  &size  *DEC  (2 0)`
   d. `DCL  &width  *DEC  VALUE(154.6789)`
   e. `DCL  &height  *DEC  VALUE(1)`
   f. `DCL  &city  *CHAR  VALUE('Memphis')`
   g. `DCL  &state  *CHAR`
   h. `DCL  &phone  *CHAR  15  VALUE('555-1212')`
   i. `DCL  &tax_rate  *DEC  (4 3)  VALUE(3.625)`

3. Consider the following variable declarations. Which are valid declarations? Which are invalid? For those that are invalid, explain why.

   a. `DCL  &name`
   b. `DCL  &first_name  3  *CHAR  VALUE('Bob')`
   c. `DCL  VAR(&lastname)  LEN(5)  TYPE(*CHAR)  VALUE('Jones')`
   d. `DCL  &city  *CHAR  (10 0)`
   e. `DCL  &State  *CHAR  VALUE('Mississippi')`
   f. `DCL  &countryname  *CHAR  VALUE('USA')`
   g. `DCL  &number  *DEC  VALUE(6.123456)`
   h. `DCL  &NEXT  *DEC  VALUE(234.9645375676)`
   i. `DCL  &NAME2  *CHAR  VALUE(1623)`
   j. `DCL  &NAME3  *CHAR  VALUE(Robert Frost)`
   k. `DCL  &switch16  *LGL  LEN(1)  VALUE('0')`
   l. `DCL  &switch65  *LGL  VALUE(1)`
   m. `DCL  &Address_1  *CHAR  10000`
   n. `DCL  &zip_code  *DEC  (17 9)`

4. Which of the following are invalid character (*CHAR) type values? Why? How would you correct them?

   a. `VALUE(12345)`
   b. `VALUE(-12345)`
   c. `VALUE(X'ABCD')`
   d. `VALUE('Today is Sunday')`
   e. `VALUE(Today is Monday)`
   f. `VALUE(today)`
   g. `VALUE('*$%@)(@#PRT123')`
   h. `VALUE(SIXTY)`
   i. `VALUE(*hjert)`

5. Which of the following are invalid decimal (*DEC) type values? Why? How would you correct them?

   a. `VALUE(Ø)`
   b. `VALUE(123A)`
   c. `VALUE(-476.8)`
   d. `VALUE(1776)`
   e. `VALUE(John Doe)`
   f. `VALUE('123.99')`
   g. `VALUE($2.99)`
   h. `VALUE(3.14+)`
   i. `VALUE('Ø')`

6. Which of the following are invalid logical (*LGL) type values? Why? How would you correct them?

   a. `VALUE(Ø)`
   b. `VALUE('1')`
   c. `VALUE(*true)`
   d. `VALUE(*on)`
   e. `VALUE('false')`
   f. `VALUE('X'Ø'')`

7. Which of the following are invalid integer (*INT) type values? Why? How would you correct them?

   a. `VALUE(Ø)`
   b. `VALUE('2')`
   c. `VALUE(3.14)`
   d. `VALUE(456)`
   e. `VALUE('- 45')`
   f. `VALUE('X'FD'')`

8. Which of the following are invalid integer (*UINT) type values? Why? How would you correct them?

   a. `VALUE(56)`
   b. `VALUE('-2')`
   c. `VALUE(3.14)`
   d. `VALUE(456)`
   e. `VALUE('655')`
   f. `VALUE('X'FD'')`

Programming Assignments

Note

- **In the following assignments, *yourLib* refers to the library assigned to you by your instructor.**
- **If you do not have a source physical file named QCLLESRC in *yourLib* for CL source members, ask your instructor to help you create one.**
- **The internal comments shown below are required for all program assignments in this book. Replace the *italic text* shown here with the proper information.**

```
0000.01 /*============================================================*/
0001.00 /* Program Name: (Program name)                               */
0002.00 /* Author:       (Your name)                                  */
0003.00 /* Date Completed: (The current date)                         */
0004.00 /* Chapter: xx Assignmant xx                                  */
0004.01 /* Program Description: (Program description)                 */
0004.02 /*                                                            */
0004.03 /*                                                            */
0004.04 /*============================================================*/
```

- **The program naming conventions used in this book might not be the same as those your instructor would like you to use. Consult your instructor for the standards you should use.**
- **Ask your instructor if you should use the generic program that came with this book as a template.**
- **Refer to Appendix A to learn the meanings of the commands in the assignments below.**
- **To accomplish the exercises, you may need to consult Appendix B for details about programming tools.**

1. In this assignment, you practice declaring variables for use in a CL program. The program sends you a message when it is done. Add a member named SA0701*XXX* to source physical file QCLLESRC in *yourLib*, substituting your initials for *XXX*.

   a. Add the program information section and the program linkage section to the member.
   b. Declare the following variables for use in the program:

      - A character variable named Time, six positions long, with an initial value of blanks.
      - A character variable named User, 10 positions long, with an initial value of your system profile.
      - A character variable named Customer, seven positions long, with an initial value of 55345.
      - A character variable named Program, nine positions long, with an initial value of the program name.
      - A decimal variable named Taxrate, five positions long, four of which are to the right of the decimal point. Give the variable an initial value of 7.1625.
      - A logical variable named Errorflag, initially false.

   c. Add SNDPGMMSG statements in the procedure section to display these variables: Customer, User, and Program.
   d. Include the following statements in the procedure section:

      - `SNDPGMMSG ('The program has completed')`
      - `RETURN`

   e. Compile the program.
   f. Examine the variable cross-reference in the compile listing to see whether your variable declarations are correct.
   g. Run the program. What is the result?

Note

You should notice several errors in the compile listing. These errors are mostly CPD0726, telling you that you have declared variables within the program but have not referred to them. Notice that these errors have a severity level of 10. Because the errors have a severity level less than 30, they do not stop the program from being compiled successfully. They are simply there to warn you that you might have a problem because normally you would not declare variables that you don't use.

The other warning is CPD0791. This message, which has a severity level of 00, appears if you do not use labels within your program. It is not necessarily an error, but the compiler wants you to know there could be a problem.

2. In this assignment, you practice using the DCLF command and declaring variables in a CL program. The program sends you a message when it is done. Add a member named SA0702*XXX* to source physical file QCLLESRC in *yourLib*, substituting your initials for *XXX*.

   a. Add the program information section and the program linkage section to the member.
   b. Declare the following variables for use in the program:

      - A character variable named User, 10 positions long, with an initial value of your system profile.
      - A character variable named Program, nine positions long, with an initial value of the program name.
      - A logical variable named Errorflag, initially false.

   c. Open the file GTCSTP in *yourLib* using the DCLF command. (*Note:* In Chapter 6, Programming Assignment 4, you were asked to write a program to copy file GTCSTP to *yourLib*.)
   d. Add SNDPGMMSG statements in the procedure section to display these variables: User and Program.
   e. Include the following statements in the procedure section:

      - `SNDPGMMSG ('I've opened one file in this program')`
      - `RETURN`

   f. Compile the program.
   g. Examine the variable cross-reference in the compile listing to see whether your variable declarations are correct.
   h. Run the program. What is the result?

Note

You should notice several errors in the compile listing. These errors are mostly CPD0726, telling you that you have declared variables within the program but have not referred to them. Notice that these errors have a severity level of 10. Because the errors have a severity level less than 30, they do not stop the program from being compiled successfully. They are simply there to warn you that you might have a problem because normally you would not declare variables that you don't use.

The other warning is CPD0791. This message, which has a severity level of 00, appears if you do not use labels within your program. It is not necessarily an error, but the compiler wants you to know there could be a problem.

3. In this assignment, you practice using the DCLF command in a CL program. The program sends you a message when it is done. Add a member named SA0703*XXX* to source physical file QCLLESRC in *yourLib*, substituting your initials for *XXX*.

   a. Add the program information section and the program linkage section to the member.
   b. Open the following files in the program. Will you need to use the OPNID keyword? If so, discuss with your instructor the naming convention.

      i. Open the file GTCSTP in *yourLib* using the DCLF command. (*Note:* In Chapter 6, Programming Assignment 4, you were asked to write a program to copy file GTCSTP to *yourLib*.)
      ii. Open the file GTCSTP in your library list using the DCLF command.
      iii. Open the display file SA0703D. Ask your instructor where this display file is located.

   c. Include the following statements in the procedure section:

      - `SNDPGMMSG ('I've opened three file instances in this program.')`
      - `RETURN`

   d. Compile the program.
   e. Examine the variable cross-reference in the compile listing to see whether your variable declarations are correct.
   f. Run the program. What is the result?

Note

You should notice several errors in the compile listing. These errors are mostly CPD0726, telling you that you have declared variables within the program but have not referred to them. Notice that these errors have a severity level of 10. Because the errors have a severity level less than 30, they do not stop the program from being compiled successfully. They are simply there to warn you that you might have a problem because normally you would not declare variables that you don't use.

The other warning is CPD0791. This message, which has a severity level of 00, appears if you do not use labels within your program. It is not necessarily an error, but the compiler wants you to know there could be a problem.