CHAPTER 2

Getting Started

2.1. Chapter Overview

This chapter introduces you to the sections of an ILE RPG program. It also illustrates how to write a simple file read/write program by using a procedural approach, as well as how to include comments within your programs as documentation.

2.2. The Sections of an ILE RPG Program

ILE RPG programs consist of four main sections:

- **Control options** section—provides default options for the program
- **Declarations** section—identifies and defines the files, variables, and other data items a program is to use
- **Main procedure** section—details the processes, calculations, and procedures the program is to perform
- **Subprocedure** section—includes declarations and processes for optional distinct functions (subprocedures) of the RPG program that either the main procedure section or other subprocedures can execute once or many times

Not every program includes each section. Within your source code, though, these sections must appear in the order above with all program lines in the same section grouped together. Additionally, within the declarations section, good programming style dictates that you group certain types of declarations (e.g., file declarations) and code them in a consistent order.

Note

RPG is a rapidly evolving language. This textbook reflects ILE RPG's capabilities at Release 7.2. This release introduced significant changes to the language, including greatly expanded free-format syntax.

Wherever practical, the text is applicable to all officially supported release levels, including future ones. But if a feature requires a specific level, we try to mention that requirement.

The ILE RPG compiler processes source member entries in columns 6–80 of each line. Columns 1–5 and those beyond column 80 are not used. You code free-format RPG statements in columns 8–80 of each line in a source member. Column 6–7 *must* be blank. If a source line contains an entry in columns 6–7, the compiler assumes that the line uses an older fixed-format specification. Free-format statements generally consist of an instruction that indicates the purpose of the statement, followed by zero or more **keywords** and values that further refine the instruction. For example, the following statement defines a variable named `Today`:

```
Dcl-s Today Date(*Iso) Inz(*Sys);     // Today's date
```

This statement uses the `Dcl-s` (Declare Standalone Variable) instruction to indicate the purpose of the statement. It then names the variable (`Today`) and uses the `Date` keyword to designate the data type of the variable, along with the display format of the date (`*ISO`). The `Inz(*Sys)` keyword initializes the variable with the current system date when the program starts; that is, the starting value of the `Today` variable is the current system date.

It's important to notice the semicolon (;) at the end of the statement. Free-format RPG uses the semicolon as a **terminator character** (similar to the way a period ends a sentence). A single statement can span multiple lines of code, but only one statement can appear on any single line.

You can add **comments** to a program to document it for other programmers who might maintain the program later, or to explain it to yourself. A comment begins with two slashes (//). When the compiler encounters two slashes, it treats the rest of the line as a comment. A line can also consist solely of a comment with no other instructions.

Tip

As you begin to work with RPG statements, don't be overwhelmed by what appear to be hundreds of entries with multiple options. Fortunately, many entries are optional, and you use them only for complex processing or to achieve specific effects. This book introduces these entries gradually, initially showing you just those entries needed to write simple programs. As your mastery of the language grows, you will learn how to use additional entries required for more complex programs.

2.3. A Sample ILE RPG Program

Let's start with the minimal entries needed to procedurally code a simple read/write program. To help you understand how to write such a program, we walk you through the process of writing an RPG program to solve the following problem.

You have a file, Customers, with records (rows) laid out, as in Figure 2.1. This layout, called a **record format**, is stored in the Customers file itself when the file is created. The record format describes the **fields** (columns) in a record. Every record format has a name (CUSTSREC in Figure 2.1). ILE RPG requires that the record format name be distinct from the filename. When the RPG program refers to the Custrec format, it uses the layout in Figure 2.1. Chapter 3 further explains how to create the Customers file by using SQL.

Properties | Source Prompter | Field Table | Commands Log | Data Table

: File FLIGHTPROJ/CUSTOMERS (9 Fields)

Name	Record	Type	Length	Text
CUSTNO	CUSTSREC	Character	9	CUSTOMER NUMBER
CFNAME	CUSTSREC	Character	15	CUSTOMER FIRST NAME
CLNAME	CUSTSREC	Character	20	CUSTOMER LAST NAME
CADDR	CUSTSREC	Character	30	CUSTOMER STREET ADDRESS
CZIP	CUSTSREC	Character	5	ZIP CODE
CPHONE	CUSTSREC	Character	10	CUSTOMER PHONE
CEMAIL	CUSTSREC	Character	50	CUSTOMER EMAIL
CDOB	CUSTSREC	Packed Decimal	8.0	CUSTOMER DATE OF BIRTH
CGENDER	CUSTSREC	Character	1	F=FEMALE M=MALE

Figure 2.1: Record layout for Customers file

You want to produce a report laid out according to the example in Figure 2.2. This report layout is also represented in a special kind of file called a **printer file**. For this example, you call the printer file Custlist. Like the Customers file, the Custlist file contains record formats that describe the lines to print on the report. When the output is a printed report rather than a data file, *record* roughly translates to one or more related *report lines*. Most reports include several different report-line formats. In the example, name the record formats Header, Detail, and Total. The Header format describes lines 1–9 in Figure 2.2, and the Detail format

describes line 10 and subsequent lines until the Total format, which prints at the end of the report. Chapter 3 details how to create the printer file and its formats by using a utility called *Data Description Specifications (DDS)*.

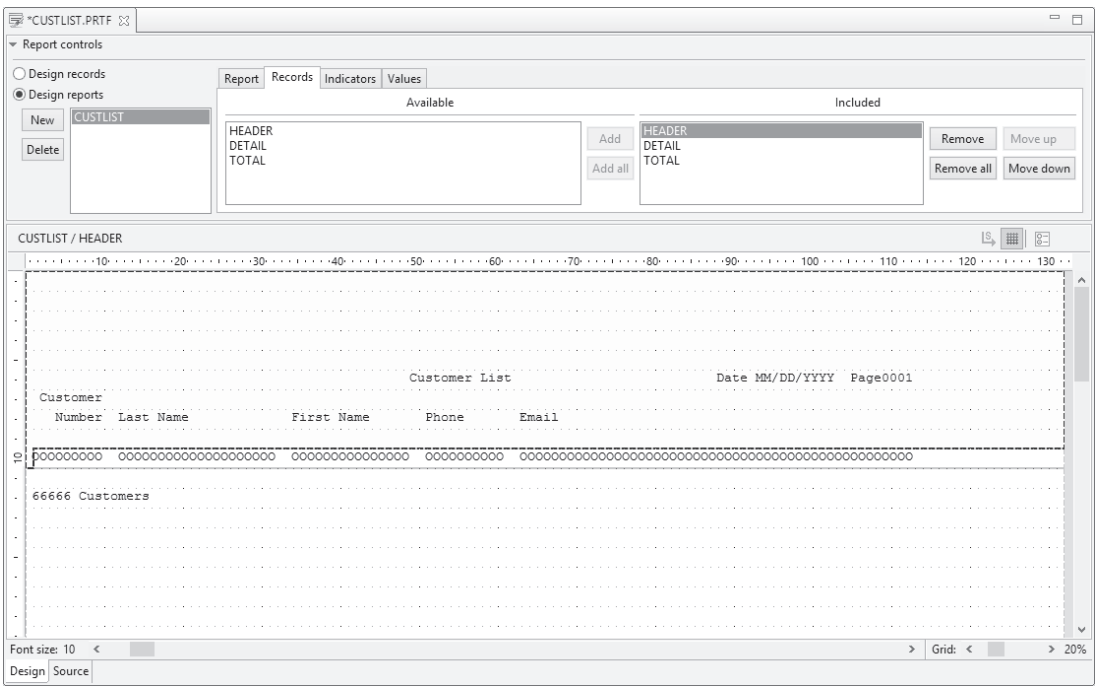

Figure 2.2: Report layout for Printer file

When you compare the desired output with the Custsrec record layout, you can see that the Detail format's output fields are present on the input records, but not all input fields in Customers are used in the report. The processing consists of reading each record from the input file, counting it, writing that data to the report with appropriate headings, and formatting the variable data. Finally, at the end of the report, the Total format prints the record count.

The following is the completed sample ILE RPG program. Note that the order of the program statements is control options, declarations, and main procedure. (This program does not include subprocedures.) RPG requires this order. Also note that you can use blank comment lines or comment lines of dashes to visually break the program into logical units and that using lowercase lettering within internal documentation helps it stand out from program code.

```
// ------------------------------------------------------------------
// This program produces a customer listing report. The report data
// comes directly from input file Customers.
//
                                                          Continued
```

```
//      Date Written:  10/31/2013, Bryan Meyers
// ------------------------------------------------------------------

// ---------------------------------------------------- Control options
Ctl-opt Option(*Nodebugio);

// ---------------------------------------------------- File declarations
Dcl-f Customers Disk    Usage(*Input);
Dcl-f Custlist  Printer Usage(*Output) Oflind(Endofpage);

// ---------------------------------- Standalone variable declarations
Dcl-s Endofpage Ind Inz(*On);

// ------------------------------------------------------ Main procedure
Read Customers;

Dow Not %Eof(Customers);

  If Endofpage;
    Write Header;         // Header is a format in the Custlist file
    Endofpage = *Off;
  Endif;

  Count += 1;             // Count is a variable defined in Custlist
  Write Detail;           // Detail is a format in Custlist
  Read Customers;
Enddo;

If Endofpage;
  Write Header;
Endif;

Write Total;              // Total is a format in Custlist
*Inlr = *On;
Return;
```

ILE RPG lets you use both uppercase and lowercase alphabetic characters, but the language is not **case sensitive**. Thus, any lowercase letter you use within a file or variable name is interpreted as its uppercase equivalent by the compiler. To aid in the program's readability, use **title case**, wherein each word in the source code is capitalized.

Let's examine the program, line by line, to understand its basic syntax and purpose. As we examine the program in detail, you can refer back to it to get a complete picture of the entire program. The comments each begin with // characters and require no further explanation.

Tip
It's a good idea to designate the various sections of the program with comment *divider lines*, as the example shows, to make the program easier to read and analyze.

2.3.1. Control Options

The first section, control options, is useful for controlling the program's behavior and for specifying certain compiler options. Control specifications provide the following functions:

- default formats (e.g., date formats) for the program
- changes to normal processing modes (e.g., changing the internal method the program uses to evaluate expressions)
- special options to use when compiling the program
- language enhancements that affect the entire program

A **Ctl-opt (Control option)** instruction can include more than one keyword (with at least one space between each one), and a program can have multiple Ctl-opt statements. Appendix A includes a complete list of Control option keywords and their usage. Not all programs require Ctl-opt statements, but if they are present, control options must appear as the first statements in a program.

In this example, the following instruction informs the compiler that the debugger facility (described in Appendix C) is to ignore all input and output specifications, improving the debugger's performance:

```
Ctl-opt Option(*Nodebugio);
```

Control options might also dictate the date and time formats to use, as in the following example:

```
Ctl-opt Datfmt(*USA) Timfmt(*HMS);
```

2.3.2. Declarations

Next is the declarations section. RPG uses this section to declare (define) all the data items that the program needs to do its job. Every data item the program requires must be defined

to the program. The example program includes two types of declarations: file declarations and standalone variable declarations. The declarations can be in any order, but it's a good idea to group similar declarations and organize them in a logical fashion. Let's start with file declarations.

2.3.2.1. File Declarations

File declarations describe the files your program uses and define how to use the files within the program. Each file a program uses requires its own file declaration, which begins with a **Dcl-f (Declare File)** instruction. Although you can declare the files in any order, it is customary to declare the input files first:

```
Dcl-f Customers Disk    Usage(*Input);
Dcl-f Custlist  Printer Usage(*Output) Oflind(Endofpage);
```

When a program declares a file, all the record formats and fields in that file become available to the program. No further declarations are necessary for the program to use those items. Let's examine each of the entries for the sample program. As you continue in subsequent chapters, we'll explain the entries not described here. (Appendix A includes a complete summary of all the ILE RPG instructions.)

In the illustrative problem, file Customers contains the data you want to process. The program's output is a printed report. Although you usually think of a report as hardcopy, rather than as a file per se, in RPG you produce a report through a printer file. This file then resides as a spooled file in an output queue, where it waits until you release it to a printer. Your administrator or instructor will tell you which output queue to use for your programs and explain how to work with spooled files in the output queue.

2.3.2.1.1. Filename

The first entry following the Dcl-f instruction names the file. In ILE RPG, filenames can be a maximum of 10 characters. They must begin with an alphabetic character or one of the special characters $, #, or @. The remaining characters can be alphabetic characters, numbers, or any of the four special characters _, $, #, and @. A filename cannot contain blanks embedded within the permissible characters.

The practice problem's input file is called Customers. The report file is Custlist.

2.3.2.1.2. Device

The entry following the filename indicates the device associated with a file. Database files are stored on disk. Accordingly, Disk is the appropriate device entry for the Customers file. The device associated with printer files is Printer.

2.3.2.1.3. File Usage

The Usage keyword specifies how the program is to use the file. The two types in this program are *Input and *Output. An **input file** contains data to be read by the program, and an **output file** is the destination for writing output results from the program. In the example, Customers is an input file, and Custlist is an output file.

2.3.2.1.4. Overflow Indicator

RPG supports many other file keywords (discussed in Chapter 4, and listed in Appendix A) to give you an opportunity to amplify and specialize the basic file description. Typically, you code them with one or more values (arguments) in parentheses immediately following the keyword itself. You can code more than one keyword on a declaration line. If a declaration requires more than one line, you can simply continue coding keywords on subsequent lines, ending the last line with a semicolon (;).

In the sample program, the printer file uses one such keyword: **Oflind** (Overflow indicator). **Overflow** is the name given to the condition that happens when a printed report reaches the bottom of a page. Usually, when overflow occurs, you eject the printer to the next page and print a new set of heading lines before printing the next detail line. Your program can automatically detect overflow through the use of a variable called an *overflow indicator*. The Oflind keyword associates the printer device with the overflow indicator for that file. In the example, the overflow indicator is named Endofpage (you can name it anything you choose). If the Custlist file signals overflow, the Endofpage indicator is automatically set to *On. You can then test that indicator just before printing a detail line to determine whether you want to print headings first. After printing the headings, the program should set Endofpage to *Off and then wait for the printer file to turn it *On again.

No other Dcl-f entries are required to describe the files the sample program used. In this introductory explanation, we skip some of the entries that are not needed in this program (we'll cover them later).

2.3.2.2. Standalone Variable Declarations

Variable declarations describe those variables that do not originate from a file and that do not depend upon a specific structure or format. They *stand alone* in the program and are often referred to as **standalone variables**. For these data items, the program needs to know, at a minimum, the name of the variable and its data type (e.g., character or numeric). Additional keywords can amplify or specialize the variable's properties. The example program declares one variable by using the **Dcl-s (Declare Standalone Variable)** instruction:

```
Dcl-s Endofpage Ind Inz(*On);
```

2.3.2.2.1. Variable Name

The first entry following the Dcl-s instruction names the variable. In ILE RPG, variable names can be up to 4,096 characters (although the practical limit is much lower). They must begin with an alphabetic character or one of the special character $, #, or @. The remaining characters can be alphabetic characters, numbers, or any of the four special characters _, #, $, and @. A variable name cannot contain blanks embedded within the permissible characters.

The example declaration describes the Endofpage variable, which you are using as the overflow indicator for the Custlist file. Because this variable isn't defined anywhere else, you must declare it here.

Tip

Although RPG allows them, avoid the use of special characters $, #, or @ in RPG names. These special characters may not exist in all languages and character sets within which your program may attempt to compile. When the language or character set cannot recognize the character, the compiler cannot successfully translate the code. Also avoid the underscore (_) in an RPG name; it's a *noisy* character and doesn't significantly aid the readability of your program.

Although not an RPG requirement, it is good programming practice to choose field names that reflect the data they represent. For example, Loannumber is far superior to X for the name of a field that stores loan numbers. Choosing descriptive field names can prevent your accidental use of the wrong field as you write your program, and it can help clarify your program's processing to others who may have to modify the program.

If the name won't fit on a single line, consider renaming the data item. You can, however, use an ellipsis (…, three periods across three positions) as a special continuation character within the name to allow a longer name. On the following lines, you simply continue the definition.

2.3.2.2.2. Data Type

Following the variable name, you use a keyword to indicate the general type of data the variable represents (e.g., character, numeric, date) as well as how the program is to store it internally. Chapter 4 examines the various data types that RPG supports. In the example, the Ind data type designates the variable as an indicator. An **indicator** (which many other computer languages refer to as a **Boolean data type)** is a single-byte variable that can contain only two logical values: '1' or '0'. You can also refer to these values by using the figurative constants *On and *Off, respectively. Indicator data is usually used within an RPG program to signal a true/false condition.

In the example, Endofpage is an indicator that the program uses to signal printer overflow.

2.3.2.2.3. Initialize Keyword

You can enter the remaining keywords in a variable declaration in any order. In the example, you have only one additional keyword: **Inz (Initialize)**. The purpose of a variable is to hold a value. Once you've defined a standalone variable, you can assign it a value, use it with operations, or print it. The Inz keyword supplies an **initial value** for a variable. This is the value the variable has when the program first starts. In the example, you initialize Endofpage to have a value of *On.

2.3.3. Main Procedure Section

You have now defined the files and variables your application is to use. Next, you need to describe the processing steps to obtain the input and write the report. That is the purpose of the **main procedure**. The main procedure is the main body of your program—the part that outlines the processes, calculations, and procedures that your program executes. In many cases (as in the example), the program does all the work in the main procedure. The main procedure is the first part of your program that is executed when you initially call it.

In the example, the main procedure is coded with no special designation and without an explicit name. It simply consists of a number of instructions that the program executes to do its work. When the program is called, an internal sequence of preset events known as the **RPG cycle** starts the program, initializes its storage, and executes the main procedure. When the main procedure is finished, the cycle shuts down the program, deallocates all objects it is using, and cleans up its storage. Because the built-in cycle controls the execution of the main procedure, this type of program is termed a **cycle main program**. Unlike with some other computer languages, an RPG cycle main program does not require that you explicitly name the main procedure. Most of the programs you encounter will be cycle main programs. Another ILE RPG program model, the linear main program, which Chapter 14 covers, does allow you to explicitly name the main procedure.

Before coding the main procedure, you need to develop the logic to produce the desired output. Generally, you complete this stage of the program development cycle—designing the solution—before doing any program coding, but we delayed program design to give you a taste of the language.

You can sketch out the processing of your program by using **pseudocode**, which is simply stylized English that details the underlying logic needed for a program. Although no single standard exists for the format used with pseudocode, it consists of key control words and indentation to show the logic structures' scope of control. It is always a good idea to work out the design of your program before actually coding it in RPG (or in any other language).

Pseudocode is language independent and lets you focus on what needs to be done, rather than on the specific syntax requirements of a programming language.

The program exemplifies a simple read/write program in which you want to read a record, increment the record count, process that record, and repeat the process until no more records exist in the file (a condition called **end-of-file**). This kind of application is termed **batch processing** because once the program begins, a *batch* of data (accumulated in a file) directs its execution. Batch programs can run unattended because they do not need control or instructions from a user.

The logic the read/write program requires is quite simple:

Correct algorithm

Read a record

While there are records

    Print headings if necessary

    Increment the record count

    Write a detail line

    Read the next record

Endwhile

Print last report total line

End program

Note that While indicates a repeated process, or loop. Within the loop, the processing requirements for a single record (in this case, simply writing a report line) are detailed and then the next record is read.

You may wonder why the pseudocode contains two read statements. Why can't there be just a single read, as in the first step within the following While loop?

Incorrect algorithm

While there are records

    Read the next record

    Print headings if necessary

Increment the record count

Write a detail line

Endwhile

Print last report total line

End program

The preceding algorithm works fine as long as each read operation retrieves a data record from the file. The problem is that eventually the system tries to read an input record and fails because no more records exist in the file to read. When a program reaches end-of-file, it should not attempt to process more input data. The preceding incorrect algorithm will inappropriately write a detail line after reaching end-of-file.

The correct algorithm places the read statement as the last step within the While loop so that as soon as end-of-file is detected, no further writing occurs. However, if that is the only read, your algorithm will try to write the first detail line before reading any data. That's why the algorithm also requires an initial read (often called a **priming read**) just before the While loop to *prime* the processing cycle.

After you have designed the program, it is a simple matter to express that logic in a programming language—once you have learned the language's syntax. The following main procedure shows the correct algorithm expressed in ILE RPG. Notice the striking similarity to the pseudocode you sketched out earlier.

```
Read Customers;

Dow Not %Eof(Customers);

  If Endofpage;
    Write Header;
    Endofpage = *Off;
  Endif;

  Count += 1;
  Write Detail;
  Read Customers;
Enddo;
```

Continued

```
If Endofpage;
  Write Header;
Enddo;

Write Total;
*Inlr = *On;
Return;
```

The instructions usually begin with an operation that specifies an action to take. RPG supports numerous reserved words, called **operation codes**, to identify valid operations. Many of these operations are followed by operand values, which RPG calls **factors**, to provide the compiler with the details necessary to perform an operation. Other operation codes (Dow and If in this example) are followed by expressions that the program is to evaluate. Finally, each instruction must end with a semicolon (;). Spacing is not usually critical. You can code the specification in any position from 8 to 80, but positions 6 and 7 *must* be blank. You can also indent operations to clarify the flow of the program.

2.3.3.1. RPG Operations

The RPG program executes the main procedure sequentially from beginning to end, unless the computer encounters an operation that redirects flow of control. The program uses eight operation codes: Read, Dow, Enddo, If, Endif, Write, Eval, and Return. Let's look at the specific operations within the main procedure of the program. The intent here is to provide you with sufficient information to understand the basic program and to write similar programs. Several of the operations described in the following section are discussed in more detail in subsequent chapters of this book.

2.3.3.1.1. Read (Read Sequentially)

Read is an input operation that instructs the computer to retrieve the next sequential record from the named input file—in this case, your Customers file. To use the Read operation with a file, you must have defined that file as Usage(*Input). Reading a record makes all the field values in that record available to the program for processing.

2.3.3.1.2. Dow (Do While), Enddo

The Dow operation establishes a loop in RPG. An Enddo operation signals the end of the loop. Note that this Dow and Enddo correspond to the While and Endwhile statements in your pseudocode. The Dow operation repetitively executes the block of code in the loop as long as the condition associated with the Dow operation is true. Because the program's Dow condition is preceded by the word Not, this line reads, "Do while the end-of-file condition

is not true." It is the direct equivalent of the pseudocode statement "While there are more records…," because the end-of-file condition turns on only when your Read operation runs out of records.

The %Eof entry in this statement is an ILE RPG **built-in function**, which returns a true ('1' or *On) or false ('0' or *Off) value to indicate whether or not the file operation encountered end-of-file. Built-in functions (or BIFs) perform specific operations and then return a value to the expression in which they are coded. Most BIFs let you enter values called **arguments** in parentheses immediately following the BIF to govern the function. In this case, %Eof(Customers) means that you want your program to check the end-of-file condition specifically for the Customers file.

The **Enddo** operation marks the end of the scope of a Do operation, such as Dow. All the program statements between the Dow operation and its associated Enddo are repeated as long as the Dow condition is true. Every Dow operation requires a corresponding Enddo operation to close the loop.

2.3.3.1.3. If

RPG's primary decision operation is the **If** operation. When the relationship expressed in the conditional expression coded with the If operation is true, all the calculations between the If and its associated Endif operation are executed. However, when the relationship is not true, those statements are bypassed. By coding

```
If Endofpage = *On;
```

or simply (because Endofpage is an indicator)

```
If Endofpage;
```

you are instructing the program to execute the subsequent lines of code only if the overflow indicator Endofpage is *On:

```
Write Header;
Endofpage = *Off;
```

The **Endif** operation marks the end of an If operation's scope. All the program statements between the If operation and its associated Endif are executed as long as the If condition is true. Every If operation requires a corresponding Endif operation to close the block.

Tip

It is common practice to indent blocks of code that appear between Dow or If and their associated Enddo or Endif operations. By indenting the blocks, you can easily see which code is associated with the Dow or If operation. Don't overdo it, though. Indenting a couple of spaces is enough.

2.3.3.1.4. Write (Write a Record)

A **Write** operation directs the program to output a record to an output file. In the example, because the output file is a printer file, writing a record to the file has the effect of printing the format, consisting of one or more lines. The first Write operation specifies Header as the record format to print if Endofpage has been reached. As a result, the three heading lines of your report are printed. Later, a second Write specifies the Detail format. When the program executes this line of code, the record format Detail is printed, using the values of the fields from the currently retrieved Customers record. The last Write operation prints the Total record format. Header, Detail, and Total are record formats that reside in the Custlist file (Figure 2.2). All the information you need to format the printed report is in the file, not in the RPG program.

```
Write Header;
Write Detail;
Write Total;
```

2.3.3.1.5. Eval (Evaluate Expression)

The **Eval** operation assigns a value to a variable. In the sample program, coding

```
Eval Endofpage = *Off;
```

assigns the value *Off to the overflow indicator Endofpage. You do this after printing the heading lines so that the program knows that it is no longer necessary to print the headings until Endofpage is reset to *On automatically. (Endofpage is initially set to *On to ensure that the program prints headings on the first page of the report.)

In most cases, specifying Eval is optional. You can simply code the assignment expression without explicitly coding the Eval operation:

```
Endofpage = *Off;
```

Later in the program, you use the line

```
*Inlr = *On;
```

to assign the value *On to a special reserved indicator variable called **Last Record** (coded as *Inlr, read as indicator LR). *Inlr (commonly referred to simply as LR) performs a special function within ILE RPG. If LR is on when the program ends, it signals the computer to close the files and free the memory associated with the program. If LR is not on, the program continues to tie up some of the system's resources even though the program is no longer running.

You may have also noticed this implied Eval statement in the example:

```
Count += 1;
```

This statement uses the += operator, one of several **concise operators** that RPG supports. This statement simply adds 1 to the current value of the Count variable. It is equivalent to coding the following:

```
Count = Count + (1);
```

RPG's concise operators are simply shortcut coding techniques you can use when the result of an expression (Count, in this case) is also the first operand in the expression. RPG supports the following concise operators:

- += to increment a variable
- -= to decrement a variable
- *= for multiplication
- /= for division
- **= for exponentiation

2.3.3.1.6. Return (Return to Caller)

The **Return** operation returns control to the program that called it—either the computer's operating system or perhaps another program. Program execution stops when a Return is encountered. Although your program ends correctly without this instruction (provided you have turned on LR), including it is a good practice. Return clearly signals the endpoint of your program and lets the program become part of an application system of called programs. (Chapter 13 deals with called programs in detail.)

2.4. Building the Program

To create this program, you use an editor, such as SEU or LPEX, to enter the ILE RPG code into a source file member with a member type of RPGLE. A source file **member** is a set of data within a source file. Usually, a source file contains more than one member, each of which holds the source code for a single program. Once the source member contains all the necessary code, you then use the **CRTBNDRPG (Create Bound RPG Program)** command to compile the source and create the program. Figure 2.3 shows the prompted CRTBNDRPG command. The compiler generates a listing, which you use to verify the creation of the program or to find errors. Appendix B describes the tools you might use to edit and compile a program.

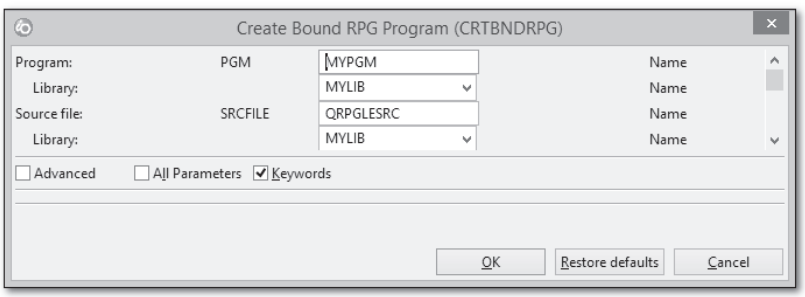

Figure 2.3: CRTBNDRPG command

Once the program is successfully created, execute it by using the CALL command. The output of your program is a spooled file, CUSTLIST, in an output queue. You can view the report by using the WRKSPLF (Work with Spooled Files) command, or you can release it to a printer.

2.5. Navigating Legacy Code

Earlier releases of RPG (Release 7.1 and earlier) identify the sections of a program by using different kinds of lines, called **fixed-format** specifications. Fixed format means that the location of an entry within a program line is critical to the entry's interpretation by the RPG compiler. Although you can write most modern programs by using **free-format** coding, which does not have significant positional restrictions, you may encounter fixed format when you are maintaining existing programs. Good programming style prefers free-format coding, but you can freely mix fixed format and free format within the same program.

Each type of fixed-format specification has a particular purpose. Following are the specification types:

- Header (Control) specifications—provide default options for the source
- File specifications—identify the files a program is to use
- Definition specifications—define variables and other data items the program is to use
- Input specifications—depict the record layout for program-described input files

- Calculation specifications—detail the procedure the program is to perform
- Output specifications—describe the program output (results)
- Procedure boundary specifications—segment the source into units of work, called *procedures*

Not all programs need every kind of fixed-format specification. Fixed-format specification types require a different identifier, or form type, which must appear in position 6 of each program line. A File specification line of code, for example, must include an F in position 6. For this reason, File specifications are commonly called **F-specs**.

Fixed-format specifications must appear in a specific order, or sequence, within your source code, with all program lines that represent the same kind of specification grouped together. The following list illustrates the order in which the specifications are grouped. Notice that the fixed-format specifications generally follow the same order as the free-format sections of a program.

H	Header (control) specifications
F	File specifications*
D	Definition specifications*
I	Input specifications
C	Calculation specifications
O	Output specifications
P	Procedure boundary
F	File specifications for procedure*
D	Definition specifications for procedure*
C	Calculation specifications for procedure
P	Procedure boundary

** Beginning with Release 7.2, you can mix File specifications and Definition specifications in any order.*

Most specifications require fixed-position entries in at least part of the specification. The editor you use to enter your source code can provide you with prompts to facilitate making your entries in the proper location. (Appendix B provides more information about editors.)

Even though specifications are primarily fixed format, some also support a free-form area, where you can code keywords and values with little or no regard to their specific location within the free-form portion of the specification. Beginning with Version 5, RPG supports free-format calculation specifications, but—until Release 7.2—all other specifications must be fixed format.

Note
The fixed-format code samples in this book use two (or more) ruler lines to help you determine where to insert your fixed-format entries. The first ruler line indicates column position, and the following line (or lines) contains *prompts* similar to those an editor supplies. Most editors also provide a similar ruler line near the top of the editing window. These ruler lines should not appear in your source code; they are provided to help you understand where entries should appear.

When you begin maintaining a fixed-format program, notice that an entry does not always occupy all the positions allocated for it within a specification. When that happens, a good rule is that alphabetic entries start at the leftmost position of the allocated space, and unused positions are to the right. Numeric entries, however, are usually right-adjusted, and unused positions are to the left.

2.5.1. Fixed-Format Specifications for a Sample Program

Let's look at a fixed-format version of the example program and compare it with the free-format version shown earlier.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+ ... 8
 * ----------------------------------------------------------------------
 * This program produces a customer listing report. The report data
 * comes directly from input file Customers.
 *      Date Written:  10/31/2013, Bryan Meyers
 * ----------------------------------------------------------------------

 * ------------------------------------------------------- Control options
H Option(*Nodebugio)

 * ------------------------------------------------------- File declarations
FCustomers IF   E            Disk
FCustlist  O    E            Printer Oflind(Endofpage)

 * ---------------------------------------- Standalone variable declarations
D Endofpage       S              N   Inz(*On)

                                                                  Continued
```

```
 * ------------------------------------------------------------ Main procedure
C                   Read      Customers

C                   Dow       Not %Eof(Customers)

C                   If        Endofpage
C                   Write     Header
C                   Eval      Endofpage = *Off
C                   Endif

C                   Eval      Count += 1
C                   Write     Detail
C                   Read      Customers
C                   Enddo

C                   If        Endofpage
C                   Write     Header
C                   Enddo

C                   Write     Total
C                   Eval      *Inlr = *On
C                   Return
```

Notice that fixed-format RPG specifications do not require a semicolon delimiter. Fixed format traditionally uses an asterisk (*) in position 7 to denote a comment. You can also use slashes (//) to begin a comment in fixed-format code, but the comment cannot be on the same line as other code; it must be on a separate line.

2.5.2. Control Specifications

Control specifications (Control options) require an H in position 6. The remaining positions, 7–80, consist of reserved **keywords**, which have special values and meanings associated with them. Fixed-format control specifications mimic the function of the Ctl-opt instruction in free-format code. The keywords have no strict positional requirements—they can appear in any order and in any position 7–80. The following header shows the layout of a Control specification for the fixed-format variation of your program:

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H Option(*Nodebugio)
```