CHAPTER 3

Using C Functions

Use library functions.
—Brian Kernighan and P.J. Plauger

Most RPG programmers don't know the C programming language and avoid technical articles or discussions that have anything to do with it. Until 1993, so did I. That year, I took some college classes in which C was the only language that could be used for assignments. I found that C was very different from RPG, but writing my own functions was very appealing. When I returned to an RPG environment in 1994, I assumed that I would not see any C programming again. In late 1994, however, IBM introduced RPG IV and ILE. At that point, integrating C functions with RPG IV programs became a possibility.

Why Use C Functions?

The RPG IV language has a rich complement of op-codes and BIFs, so why would you consider accessing functions designed for the C language? The answer is that the C language gives you some functions you don't have in RPG, as well as functions that are more efficient than native RPG operations. Integrating C functions with your normal RPG code is easy, giving you the best of both languages.

To access a C function, you need to know the function's interface, including its name, number of parameters, parameter data types, and return value, if any.

To see why C functions might be useful to RPG programming, let's look at a few examples. Let's start with the C function `Rand`, which is used for random number generation. It has to be initialized with another function, called `Srand`, which "seeds" the random number generator. The notion of seeding is to set an initial value that is kept in the system somewhere. The `Rand` function cannot be used without first doing `Srand`, which must be run only once.

Here are a few technical details to keep in mind when using these functions:

- `Srand` requires an unsigned integer parameter, and there is no return value.
- `Rand` has no parameters, and returns an unsigned integer return value.

You'll see examples that use both the `Rand` and `Srand` C functions later in this chapter.

Now, let's look at another C function, `Strtok`. This function is called *string-token*, and it is a string parser. For example, consider the string "The cat is gray." This string can be broken into separate *tokens* of "The," "cat," "is," and "gray" by using this function repetitively.

Here are few technical details to keep in mind when using this function:

- `Strtok` requires a string parameter and a delimiter character.
- The return value for `Strtok` is a pointer to a token (a character string up to the first delimiter), or a null pointer.
- It is common practice to use a blank as the delimiter, but other characters can be used.

You'll find an example that uses the `Strtok` function later in the chapter.

Now, let's consider a higher math function, `Sin` (the trigonometric function sine). Here are few technical details to keep in mind when using this function:

- The `Sin` function has one parameter, the angle in radians.
- It must be defined as a floating-point data type, either 4f or 8f.
- The return value is also floating-point.

Review the IBM reference manual *ILE C/C ++ Runtime Library Functions Reference* for details on these or any other C function you might be interested in. The document number is SC41-5607. This IBM publication can be viewed on the web at the following link:

http://publib.boulder.ibm.com/infocenter/systems/scope/i5os/topic/books/sc415607.pdf

How C Functions Work with RPG IV

Using a C function within an RPG IV program requires the use of a *prototype*. The prototype defines to RPG IV the function's return value (if any) and the parameters needed for the function.

For example, consider this use of the `Sin` function:

```
dsin                pr              8f   extproc('sin')
d angle                             8f   value
```

The prototype name is made the same as the C function. The return value definition of 8f is specified, and the external procedure name is specified (`'sin'`). On the second line, the name `angle` is optional, since names on prototype parameters are not required. The 8f is the definition of the parameter, and the keyword `value` is used, since parameters in C functions expect parameter passing by value.

The call interface is the same as for any external procedure. The `CallP` operation can be used if no return value is needed. The implicit call (which works just like a BIF) is used when the C function has a return value. As you will see throughout this book, the implicit call is emphasized as a preferred method.

Let's add a few more lines of code to the example above:

```
dSinans             s               8f
dAngleF             s               8f
dAngleP             s                7p 2   Inz(1.0456)
  /free
    AngleF = %float(AngleP);
    SinAns = sin(AngleF);
```

Three fields are defined:

- SinAns will hold the answer.
- AngleF is a floating-point work field.
- AngleP is the input angle, in packed-decimal format.

The first new line of code uses the %float BIF to convert the packed-decimal AngleP field to floating-point, and then assigns the value to the work field AngleF. The next line of code calls the function Sin implicitly, passing the work field AngleF as a parameter.

Random Numbers from C

One of the C functions that you might find useful is the random number function, mentioned earlier in this chapter. You'll see a detailed example using this function later, but for now, let me share the justification for using it in an application. The following situation occurred a couple of years ago:

I was working on an application to perform testing—giving an exam. The exam had four parts, with 10 questions in each part. The questions for each part came from a question pool, with 100 possible questions in the pool. The program needed to select, at random, 10 questions from each of the four associated question pools.

RPG IV does not have a random number function, but C does. Upon investigation, the Rand function in the C language was determined to be the proper choice. As explained earlier, Rand needs to be initialized by another C function, Srand.

The output from Rand is an integer between zero and 32,767. For the purposes of this application, I needed to scale that down to a range of zero to 99. A handy way to do this is to divide the return value by 100 (in this case), and use the remainder.

Because duplicate questions were not wanted in the exam, a check for a duplicate random number was needed, and duplicates discarded. The exam questions were stored in an array with indexes from one to 100, so the resulting random number needed to be incremented by one to match the RPG array index. Prototyping was needed for the two C functions, Rand and Srand, as follows:

```
D  Set_random           PR                  Extproc(‘srand’)

D     Seed                        10u 0

D  Get_random           PR        10u 0     Extproc(‘rand’)
```

Notice the use of the `Extproc` keyword, and the C function name in lowercase. This is required. For the `Set_random` prototype, there is no return value (the “seed” is stored somewhere in the system), but there is an unsigned integer requirement as the parameter. It is best to make this parameter as random as possible. I have found that using the micro-seconds value of the current timestamp is fairly random.

Also, ILE needs to locate the procedures `rand` and `srand`, so a binding directory needs to be specified on the H control specification. The H control specification must be placed first in the source member. Here is an example:

```
H     Bnddir(‘QC2LE’)
```

Random number setup can be done as follows:

```
D Seed                  S          10u 0
D Index                 S          10i 0
 /free
  Dou Seed = 0;
     Seed = %subdt(%timestamp( ) : *MS);
     If Seed  0;
       Set_Random(Seed);
     Endif;
  Enddo;
   // Obtaining the random number,
   //  then scaling it to 0-99 is done as follow:
  index = Get_Random( );     // index 0-32767
  index = %rem(index:100);    // index 0-99
  index += 1;               // index 1-100
```

The code above sets the seed (in the Dou group), and then uses the random number function to set an index. The index is used to access an array element from a question pool. Additional programming is needed to avoid duplicate indexes. The index is used to access a question from a question array.

Checking for duplicate random numbers and further processing of the application is not shown here.

Occasionally, the need for higher mathematical functions comes up. This might involve the use of trigonometric functions such as sine, cosine, or tangent. These are not available in RPG IV, but all of them are available in C.

The following trig functions are available in C:

- **Acos** calculates the arc cosine.
- **Asin** calculates the arc sine.
- **Atan** calculates the arc tangent.
- **Atan2** is a variation of Atan() for calculating the arc tangent.
- **Cos** calculates the cosine.
- **Cosh** calculates the hyperbolic cosine.
- **Sin** calculates the sine.
- **Sinh** calculates the hyperbolic sine.
- **Tan** calculates the tangent.
- **Tanh** calculates the hyperbolic tangent.

Using the C run-time library functions and setting up the prototypes for C functions is an easy way to use features available in the C language.

C Data Types vs. RPG IV Data Types

The data types used by C functions are nearly all integer, float, and null-terminated strings. RPG IV supports all integer and float data types, and can convert null-terminal strings to RPG character fields, and vice versa (using the %str BIF). The integer and float data types available with

RPG IV make it easy to pass variable data back and forth between RPG IV and C.

One of the hurdles for RPG IV programmers is understanding the data types used by C, and matching them to RPG IV's scheme. Typically, RPG programmers have only used zoned or packed-decimal numeric fields. Table 3.1 describes how data types in the two languages relate to each other, including recommended RPG variable sizes.

Table 3.1: Most Common Data Types for C and RPG IV

C Definition	RPG IV Prototype Parameter Definition	Notes
Int short	5i 0	Signed integer, two bytes
Int long	10i 0	Signed integer, four bytes
Unsigned int	5u 0	Unsigned integer, two bytes
Unsigned int long	10u 0	Unsigned integer, four bytes
Float	4f	Floating-point, standard precision
Double	8f	Floating-point, double precision
Char *	*	Defined in RPG with the VALUE and Options(*STRING) keywords
(*)	*	Defined in RPG with the VALUE and PROCPTR keywords

Parameter Passing to C Functions

When passing parameters to another procedure, RPG normally uses a technique called *passing by reference*, which means that a pointer address is used. In the called program, the parameter field is a template over the "passed" field in the calling program. If you modify the variable in the called program, you are really modifying the field in the calling procedure.

The C language uses a parameter passing technique called *passing by value*. In this scheme, the actual value of the parameter is passed. The value of the passed field can be changed in the called procedure, but the parameter field in the calling program is not changed. The RPG IV language can also pass parameters by value, simply by using the `VALUE` keyword for the parameter in the definition on the prototype.

A more flexible option is to use the keyword `CONST` instead of `VALUE`. This also passes parameters by value, but has some additional features. The `CONST` keyword allows you to use fields of different data types (for numeric data types) and lengths as a parameter (different than the prototype), including the use of a constant.

Character String Differences between C and RPG IV

The RPG IV language and C differ in their method of handling character strings. RPG IV uses fixed or varying-length character fields. The C language uses an array of a variable length, with the last entry in the array a null character (`hex '00'`). If you use the `Options (*String)` keyword on the parameter in the prototype for a character field, the compiler places the RPG IV character field specified into a work area, and then automatically adds the null character. The data in the work area is then passed to the C function.

Binding RPG IV and C Functions

After coding an RPG IV program with the proper prototypes, and using an implicit call to the C function using the prototype(s), there is still one more step to "glue" the pieces together. The C functions are in service programs known only to IBM, but IBM has given us access to these programs via binding directory QC2LE. By simply putting `Bnddir('QC2LE')` on the H control specification, the binder will locate and include the C functions you have requested.

You must also specify an activation group, but not the default activation group, since you are using ILE's binding by reference. While you are testing your new program, use `*New` as the activation group. This helps avoid testing problems. Otherwise, named activations stay around and are used on subsequent calls, even if you recompile and replace the program (as you will see in Chapter 5).

Here's an example of an H control specification:

```
 H  Bnddir('QC2LE') Actgrp(*New)
```

This specifies that binding directory QC2LE be used during the binding phase, and that activation group *New be applied when the program is loaded and run. When you are past the testing phase, a named activation group might be your best alternative.

Using C Functions to Make Your Job Easier

I was asked to help with a data conversion, where the customer name in the "from" database was one big field, with spaces separating the parts of the name. In the "to" database, the name was divided into title, first name, middle initial, and last name fields. Other information about the "from" database included the following:

- Not every name had a title, but if a title existed, it was a standard title.
- A middle initial was optional, but if it existed, there was a period after it.
- Every character after the middle initial was part of the last name.
- If only one name appeared, it was the last name, not the first name.

Moving each part of the multi-part name field into the right new field became a programming project. It was accomplished completely in RPG IV, but a nagging inner voice kept saying to me, "This task could have been done more easily, somehow." After some research, I found a C function that could have helped: the Strtok (string token) function.

This is a parser-type function that helps in dividing portions of a character string. Something in the string has to be the delimiter, to tell the parser where one portion ends and the next begins. The C language calls these short string portions *tokens*. For example, using the Strtok function with the sentence "See Spot Run" and a space as the delimiter, you get "See" with one pass, "Spot" with a second pass, and "Run" with a third pass. This scheme would fit my data conversion situation pretty well, where blanks would be used to separate parts of the big name field. Of course, some logic would still need to be used to place each "token" item into its rightful place.

The Strtok Function

We’ll review the most important elements of the new Strtok program in this section. (The entire program takes over a hundred lines of code, so it is not included here.) The field with the name to be parsed is FullName, defined 40A.

The first part of the main procedure is shown here:

```
h Bnddir(‘QC2LE’) Actgrp(*New)

d GetToken          pr         *     ExtProc(‘strtok’)

d  Name                        *     Value Options(*String)

d  Delimit                     *     Value Options(*String)
```

It has the H control specification with the IBM binding directory QC2LE specified, and the activation group *New. Following that is the prototype for the Strtok C function. The return value is a pointer to the found token, or a null pointer if no token is found. The two parameters are the field to be parsed and the delimiter.

The next section is a work array to hold all of the tokens, and a work field to hold the most current token:

```
d artok          s                  like(FullName) dim(30)

d token          s                  like(FullName)
```

The next few lines are work fields used in the procedure. The definition 5u 0 indicates an unsigned integer of five digits:

```
d ReturnAdr        s           *

d count            s          5u 0

d Space            c                    ' '
```

The count field defined here is used to count the number of tokens found, and used later in processing the token array elements.

The procedure begins by clearing the previous contents of the Title, FirstName, Initial, and LastName fields:

```
  /free

   //* Clear output fields

    Clear Title;

    Clear FirstName;

    Clear Initial;

    Clear LastName;
```

The procedure continues by first checking for all-blank input. Next, it left-adjusts the FullName field, and then retrieves the first token from the FullName field:

```
    If FullName = *blank;

      //  Name is not all blank

      //  Shift Name to left, removing left blanks

      FullName = %triml(Fullname);

      ReturnAdr = GetToken(FullName:Space);
```

The function call GetToken here gets the first token. In the next section, GetToken gets the remaining tokens.

The next section is a loop to obtain and store the tokens of the FullName field:

```
    Dow ReturnAdr <> *Null;  // Load Tokens into Artok array

       Token = %str(ReturnAdr);

       Count += 1;

       Artok(count) = %trim(Token);

       ReturnAdr = GetToken(*Null:Space);

    Enddo;
```

Notice that the return value for the C function is a pointer to the token. The `%str` BIF takes the null-terminated string and moves the characters into a regular character-field-token. A count is made of how many tokens are saved in the array, to handle later processing. At this point, the tokens are stored in elements of the `Artok` array.

The use of the `Strtok` C function makes parsing the long `FullName` string very easy. The remainder of the program, not shown here, just puts the tokens in the correct output fields.

Exponentiation

There was a time when RPG could do little in the way of powers and roots. Powers were done by repetitive multiplications, and the only "root" capability was the square root. The use of C functions was the only solution for anything more complex. You might have done this.

It might seem contrary to bring this up in a chapter on using C functions, but the truth of the matter is that native RPG can now provide all you need in this mathematical arena.

With the advent of the ** (exponentiation) operator, all business formulas using an exponent can be done *without* using the C function library. In RPG IV, the exponentiation operator gives you the complete capability to determine the root or power of a number.

The form of an exponentiation operation is as follows:

```
      Answer = number ** power;
```

In this formula, "power" can be a whole number, a fraction, or a mixed number. It can also have a negative sign.

Here are several examples of exponentiation, for your review:

```
   /free

    // Simple power and roots:

    Area_Circle = 3.1416 * Radius ** 2;

    //   The above computes the area of a circle using

    //   the power 2

    Cube_Root = 8 ** (1/3);

    //   The above line computes the cube root of 8,

    //   making Cube_root = 2.

    Future_Val_Annuity = Period_Amt * (((1+i)**n-1)/i);

    //   The above expression computes the future value

    //   of an annuity. i is the periodic rate, and n

    //   is the number of periods. With this formula, you can

    //   determine how much money you will have if you save

    //   Period_Amt for n periods at interest rate i.
```

As you can see, any formula needing exponents can be done without using C functions.

Summary

C functions provide capabilities that are useful to you, as an RPG IV programmer. In particular, if you have a math requirement needing the use of trigonometry or other higher mathematics, the appropriate C function will make the overall programming much easier. Other specialized C functions, such as the random number generator, are helpful for special projects.

The example of the `Strtok` C function in this chapter shows that combining C functions with native RPG operations enhances your overall programming toolset.