Moving From SabreTalk to C

Part 3: Expressions and Program Control Statements

by Jeff Robinson

In the previous installments, I discussed how SabreTalk Procedures compared to "C" functions, and the differences/similarities in data concepts and manipulation between the two languages. In this final article of the series, I'll discuss how easy it should be for SabreTalk programmers to transition to using "C" control structures and expressions.

Want to challenge your understanding and knowledge concerning SabreTalk versus "C"? Then access the online quizzes I've created on my training site at www.robisoft.com. You'll also find challenging coding tests and links to other resources that might be helpful as you learn the "C" language.

Expressions

Before we move on to programming control, let's explore the usage of expressions between "C" and SabreTalk. Expressions involve the use of operators with operands to resolve relational, logical and mathematical results in programming statements. As you transition to "C", you'll discover that the good news about expressions is that they're used almost exactly the same as you used them in SabreTalk. The main differences will be that some of the operators are slightly different, there's no restriction on when and where you can use expressions, and you can use parentheses as much as needed in order to change the operation priority.

The following tables show the corresponding use of operators from SabreTalk to "C".

SabreTalk Logical Operators	Description	"C" Equivalent
&	And	&&
|	Or	||
¬	Not	!

SabreTalk Arithmetic Operators	Description	"C" Equivalent
*	Multiplication	*
/	Division	/
+	Addition	+
-	Subtraction	-
NA	%	Modulus

SabreTalk Relational Operators	Description	"C" Equivalent
<	less than	<
>	greater than	>
=	equal to	= =
¬<	not less than	None
¬>	not greater than	None
¬=	not equal to	!=
<=	less than or equal to	<=
>=	greater than or equal to	>=

All of the equivalent “C” operands listed in the table above can be used the same way in “C” as their counterparts were in SabreTalk expressions. The only new operand seen in the table not found in SabreTalk is the modulus operator, “%”. This modulus operator works exactly like the SabreTalk “MOD” keyword in that its used to return the remainder value from a division rather than the quotient. Also to note is that, unlike SabreTalk, “C” does not support a concatenation operator like. Instead, the concatenation of strings is taken care of by a built-in “C” function namedstrcat().

Prefix & Unary Operators
Like SabreTalk, “C” supports the use of Prefix operators in the form of the logical not (!), the arithmetic plus (+), and the arithmetic minus (-). Prefix operators are used to changed the value of a single operand before the expression in which they’re in are evaluated. Additionally, “C” supports two other operators known as unary opeartors: ++ and —. These operators are used to act upon a single operand to either increment (++) or decrement (—) its value.

Other “C” Operators
The “C” language offers many other types of operators that will not be covered in this course. Included in this group are:

- Compound assignment operators
- Bitwise operators
- The conditional operator
- The comma operator

Please consult “The C Programming Language” book by Brian Kernighan and Dennis Ritchie for details on using these type of operators in “C”.

The Assignment Operator
One of the most familiar operators you’ve already seen used in this course and which you’ll be using a lot in “C” is the assignment operator, “=”. In “C”, this operator is used to to “assign” the value of a literal or variable to another variable. Assignment expression operands can be defined values, literals, variables , and structures. However, like in SabreTalk, structures can only be assigned to other structures. Now that we’ve covered the topics of expressions and assignments, we’re ready to move on to how SabreTalk and “C” compare when it comes to controlling program flow.

The IF Statement
As a SabreTalk programmer, you’re familiar with using the “IF” control statement to provide conditional execution of statements in your programs. Likewise, “C” also provides the “IF” statement as a means to evaluate expressions and to execute instructions only if the expressions tests to be “true”. In “C”, the format of the “IF” statement is:

```
if (expression)
{
 statements
}
[else
{
 statements
}]
```

During run-time, when “expression” evaluates to true, the statements following the “IF” clause will be executed; otherwise, when it evaluates to false, then they will NOT be executed; instead, the statements following the “ELSE” will be executed if the “else” clause is present. The square brackets in the preceding format indicates that the “ELSE” clause is optional.

Also, when multiple statements follow the “IF” or “ELSE” expressions, the right and left curly braces must be used to enclosed the statements in a “code block” as shown in the format. The curly braces are a way to indicate to “C” that all the statements are to executed when the associated expression evaluates to true. Unlike SabreTalk, “C” allows you to “nest” other “IF” statements. This means that you can include an unlimited number of “IF” statements within the “statements” coded in the “IF” block. Following are examples of an “IF” statement with and without the “ELSE” clause.

Example 1: the IF statement only

```
if (number1 > number2)
  number1=number2;
```

Example 2: With an ELSE clause

```
if (number1 == number2)
 number1=number1 * 2;
else
 number1=number2 * 2;
```

Example 3: With curly braces

```
if (number1 < number2)
{
 number3=number1;    /*Swap number1 and number2 values */
 number1=number2;
 number2=number3;
}
else
{
  number2=number1;  /*Make number2 the same as number1, then increment number1*/
  number1++;
}
```

While & Do Statements

In place of the “DO” statement used in SabreTalk, “C” provides the “while” and “do” statements. Like the SabreTalk “DO”, these two instructions can repetitively execute a group of statements as long as the values in the expressions controlling the loop evaluates to a true condition. The formats for these statements are:

Format 1: while statement

```
while (expression)
{
 statement block
}
```

Format 2: do-while statement

```
do
{
  statement block
}
while (expression)
```

The "(expression)" clause in both formats above is used to denote a logical operation. Like with the "IF" clause, this expression can consists of one or more operations that must resolve to a "True" or "False" condition in order for the control statement to continue looping.

The difference between the two control statements is that "while" will only execute the statements in its block if the expression is true the first time. However, the "do-while" statement will always executes its statement code block at least once before testing "expression" to see if it should stop.

As with the "if" statement format, the "statement block" clause can represent one or more actual "C" statements or even nested control statements (even other "while" and "do" statements); thus, left and right curly braces must be used to enclosed the statement block when more than one instruction is present. An example of using both statements follows:

Example: while loop

```
while (value1 < value2) /*If value1 is less than value two,*/
{
 value1++;          /*increment value1 by 1*/
 value2—;           /*decrement value2 by 1*/
}
```

Example: do-while loop

```
do                  /*Decrement value1 at least once and */
{
 value1—            /*continue to do so if its still */
}
while (value1 > value2); /*greater than value2. */
```

The "FOR" Statement

Not found in SabreTalk but provided by "C" is the "FOR" statement. The "FOR" statement provides functionality similar to the SabreTalk's "DO" statement when its used in conjunction with the "WHILE" clause. Like DO-WHILE, the "for" statement's counter value appears as the first statement in the sequence of instructions. But the "FOR" statement also provides the means to control the loop "counter" value within its clause as seen in the following format:

```
for([initial-expression];[conditional-expression];[loop-expression])
{
  statement block;
}
```

Here's an explanation of the "for" statement parts:

1. The "initial-expression" clause is used to establish the beginning value that will be used to control the "for" loop. The expression here is usually a simple assignment operation indicating what number to begin the loop counter at. The variable used in the assignment is usually also used in the "conditional-expression" and "loop-expression" clauses.

Example: for(i=0;...

2. The "conditional-expression" clause is used to determine when the "for" loop should stop executing its statement block. The for

loop will only stop executing the statement block when “conditional-expression” evaluates to False. The variable involved in this expression is usually the same one used in the “initial-expression” clause.

Example: for(i=0; i<10;...

3. The “loop-expression” is simply another assignment operation which modifies the loop control variable in order cause “conditional-expression” to eventually evaluate to False. Therefore, the variable used in this expression must somehow affect “conditional-expression” to change.

Example: for(i=0;i<10;i++)

Like the SabreTalk DO-WHILE, the “for” statement is used to execute the statement(s) in its block a specific number of times. In the example above, the “for” loop will execute 10 times before ending.

Following is a full example of a “for” loop statement. It accumulates “value1” into “result” until “value1” is no longer less than “value2”.

```
for (value1=1;value1 < value2; value1++)
{
  result=result + value1;
}
```

The Goto, Break, and Continue Statements
Additionally, “C” provides the following 3 statements to help provide further control to your program flow. Consult your “C” reference book(s) on the formats of these statements.

1. Goto statements - works the same as in SabreTalk; requires a programmer defined label as the target.
2. Break statements - within a do/while/for loop, transfers control out of the current looping statement.
3. Continue statements - within a do/while/for loop, pass controls to the next iteration of the loop.

The Switch Statement
The “switch” statement, like the “IF”, provides another means of conditionally evaluating expressions and only executing code when the expression is true. Unlike the “IF”, SWITCH statements can only test a single expression within it code body.

Although a very useful control statement, we won’t go into detail on its usage here since there is no SabreTalk equivalent. Instead, you’re encouraged study the usage of this control statement on your own and use it where you feel its appropriate.

Summary
That about concludes this article. In case you may wondering about how other SabreTalk and “C” items compare like built-in functions and macros, consult lesson #7 of my online course on this topic at www.robisoft.com. When I started this article series several issues ago, I stated that SabreTalk and “C” have more in common than they have different. Hopefully, I’ve been able to prove that up to this point. If you have any further questions on this article series, feel free to contact me at: jeff01@robisoft.com. Finally, good luck as you make the transition from SabreTalk to “C”.

Editor's Note: **After 7 years of writing articles, training material, and an impressive collection of TPF tools and utilities, our friend and colleague Jeff Robinson has announced that he will be taking a well deserved "sabbatical" to recharge his batteries, and spend more time with his family. On behalf of the staff here at ACP/TPF Today and our readers, we just wanted to say THANKS JEFF for your many contributions to the TPF community!**