Moving From SabreTalk to C

Part 2: Data Concepts and Data Manipulation

by Jeff Robinson

In the first installment, I discussed how SabreTalk Procedures compares to "C" function. In this article, I'll delve into the how the basic data elements in SabreTalk can be transformed and used in the "C" language. You can also access these lessons online at www.robisoft.com along with quizzes, tests and links to other resources to aid you in your journey from SabreTalk to "C ".

Data Concepts

SabreTalk uses the "Declare" statement to define data areas and data fields. In TPF/C, although there is no equivalent statement to the "Declare", an equivalent means of defining data areas is provided through a rich set of "data type" keywords similar to those used in SabreTalk. When defining data in SabreTalk, you used the following format (the "..." indicates multiple definitions are possible): DECLARE [*level*] variable_name [*dimension*] data type ...

Example: DECLARE item Character(500);

To define data in "C", the "data type" attribute would be indicated first in the clause; here's an abbreviated format:
Data_type variable_name [dimension], ...;

Example: char items[500];

The following table show the corresponding relationships between a "data type" in SabreTalk and those used in "C" to define data fields.

SabreTalk Data Types	Usage	C Equivalent	C Example
Character ir Char	Define character strings	char	char abc;
Binary or BIN	Define 16 or 32 bit binary data	int (short & long)	short num1; long num2;
BIT	Define number of bits	char or int data types modified by the colon operator.	unsigned char abcd : 8; unsigned int num3 : 16;
Decimal or DEC	Define decimal data items	double	double air_fare;
Decimal Float	Define decimal floating items	float	float avg_fare;

To use the "C" equivalents in a TPF/C program, place the name of the type in front of the field name (or fields) followed by a semicolon. In TPF/C, the declared field names are known as *variables* and are usually composed of alphanumeric characters and underscores.

SabreTalk also allowed for the use of multiple "dimensions" for a data field known as "arrays". "C" also allows the use of arrays and the use of subscripts to access arrays. Arrays indicate that the named data field acutally occupies contiguous areas of storage in the size denoted by the data type for the number of times specified in the array number. For example, in SabreTalk, you may have done this to indicate that a character field occupied 8 bytes of storage: DCL first_name Char(8);

In "C", you would declare the same field like this: char first_name[8];

Note that "C" uses the square brackets to indicate an array whereas SabreTalk used parentheses. Both notations indicates to the compiler that the character "string" variable actually takes up 8 bytes rather than 1 as is usual for the "Char" data type.

Initializing C Variables

In SabreTalk, you're not allowd to assign a value to your declared data areas at the time you declare them. This is not the case with "C". In "C", you can save time and code by initializing your data when you declare it. To initialize a "C" variable, simply place an "equal" sign followed by the literal data to set the variable to when you declare it.

For example:
```
short age=35,weight=175;
char first_name[8]={"Joseph"};
```

Notice also in the above example that like SabreTalk, "C" allows you to make more than one declaration in a statement. Thus, we declared both "age" and "weight" after the same data type "short". The same could have been done after the "char" data type. Also, notice that for the character "array", curly brackets had to be used to set the initialize data. This is the standard for initializing array data in "C" and must be used with any array variable that is being initialized. One reason that field names in "C" are known as variables is because the data being held in those named storage areas are subject to change. In SabreTalk, if you wanted to indicate that a declared data item could not be changed, you did something like this:

```
DCL radius DEC FLOAT CONSTANT;
CONST radius, 3.14;
```

The "Constant" keyword was coded to make the data field unchangeable. In "C", if you wanted to indicate that a field can not to be changed, you would insert the "const" data attribute keyword in front of it like this:

```
const float radius=3.14;
```

If you declared a field in the preceding manner, you could not change the initial value later in your program. Any attempts to do so in your program would result in a compiler error.

What "C" Leaves Behind

There are several data-related functions supported by SabreTalk that do not have a corresponding implementation in "C". First, unlike SabreTalk, "C" does not allow for "Picture" data attributes or "Picture" specifications when you declare your variables. Thus, there is no allowance for "Edited Character-String data", "Suppression characters", "Insertion characters", or "Drifting characters" to modify data field content. Instead, "C" allows this to be done at the coding level using special "format specifications" with functions from the standard "C" library.

Secondly, "C" does not allow you to make "label" declarations like SabreTalk. Instead, in "C" you can simply code labels starting in column 1 and follow them with the colon; doing this indicates to "C" that you want to declare a label.

Example:
```
label1: num1=num2;
```

Likewise, "C" does not allow you to "declare" function names in a DECLARE/DCL statements. (However, in "C" you must declare function prototypes before you code the function.).

Literal Data

Both SabreTalk and "C" allow for use of literal data within the source program. Literal data represents values thatyou code explicitly within your source code (i.e. no variable or symbol representation). Within "C", the use of literal data is genarally called "constant" data. The following table shows the corresponding usage of literals in both languages.

SabreTalk Literal/Constant Type	Example	"C" Constant Type	Example
Binary	25000, -6	Numeric constant	255, 255L
Bit-string	'011010'B	Numeric constant	0x80
Character-string	'hello'	Character/string constant	"hello", 'a'
Decimal	25.5	Numeric constant	25.5
Decimal Floating	28.E5	Numeric constant	3e0f, 15.75E2

As you begin coding in “C”, remember that using literals is very similar to what you did before so you can probably continue to code literals as you have always done in SabreTalk.

Pointers
Both SabreTalk and “C” make use of a data type definition known as Pointers. A pointer in “C” is actually the same as it is in SabreTalk: a variable in which an address to another location is stored. The difference to be found is in the way the two languages allow pointers to be declared. In SabreTalk, you could declare pointers by using either the “POINTER” data attribute or the “BASED” data attribute.

```
DECLARE newptr POINTER;
DECLARE secondptr BASED(newdata);
```

Both of the above declaration create pointer variables. The first is called an explicit declaration while the second is an implicit declaration. In “C”, the way to explicitly declare pointers is by using the asterisk “*” operator in a regular data definition statement. Here's an example: char *newptr;

In the preceding example, we just declared a pointer variable named “newptr” with a data type of “char”. This means that the “newptr” will hold the address of a data location containg “character-string” data. If any other data type had been used, then “C” would expect data of that type to be at the address stored into “newptr”. In “C”, there is no corresponding “BASED” keyword to implicitly declare pointers. The only implicitly declared pointeres in “C” are character arrays. Anytime you declare a character array of any non-zero size, “C” will treat the unqualified name of the character variable as a pointer. In “C”, there is no limit on the number of pointers that you can explicitly or implicitly declare. Plus, you can also declare and array of pointers to a data type. The example below declares an array of 5 pointer values for the “newptr” array.

```
short *newptr[5];
```

Initializing Pointer Variables
Initializing explicity declared pointers in SabreTalk is a process that takes place after the pointer is declared; this usually involves assigning the pointer variable another pointer’s value, using the built-in ADDR function, or using the START statement of a macro to assign the contents of a register to a pointer. In “C”, pointer variable initializaion can be done at declaration time or later within the body of the program. To initialize a pointer variable at declaration time in “C”, you must use the ampersand “&” operator to get the address of another variable — unless that variable is a character array, in which case you only have to supply the variable name. You can do either method at declaration time like this:

```
char first_name[8]={“Joseph”}, *nameptr=first_name;
short age=38, *age_ptr = &age;
```

Now, “nameptr” points to the same location as “first_name”. Notice that the unqualified usage of the “first_name” variable is treated as a pointer itself. And “age_ptr” points to the same storage location that contains the value “38”. Here, placing an ampersand in from of “age” causes its address to be assigned to “age_ptr”. There are many more details involved in the proper use of pointers in “C”. For complete information, consult “The C Programming Language” book by Brian Kernighan and Dennis Ritchie.

Structures
Another common data representation to both SabreTalk and “C” is the use of structures. Structures allow both SabreTalk and “C” to group several data fields (not necessarily the same data type) into one, cohesive logical group. In SabreTalk, you did this by coding something like the following:

```
DCL 1 address,
        2 street CHAR(20),
        2 city CHAR(20),
        2 state CHAR(10),
        2 zip PIC ‘99999’;
```

However, in “C”, the *struct* keyword is used to indicate the start of a structure. Also, the data feilds declared within the structure must be enclosed in an opening/closing curly brace pair and the use of the “level” qualifier numerals (i.e., “1”, “2”) are not allowed. Here’s how the preceding SabreTalk structure would look in “C”:

```
struct address
{
        char street[20];
        char city[20];
        char state[10];
        int zip;
};
```

In the structure above, “address” is simply the name we’ll give to this structure definition to identify it later. Declaring the structure as we’ve just done doesn’t cause any storage allocation to take place.

Notice that the semicolon in the “C” version follows each declared data field (instead of a comma like in SabreTalk) and the structure’s closing brace.
In “C”, you can have structures imbedded within structures; this feature corresponds to the SabreTalk feature of having multiple levels within a structure.

Allocating Structures
In “C”, structure declarations in and of themselves don’t allocate any storage. Storage is only allocated when new variables are defined as belonging to the type defined by a previously declared structure. One way to assign variables to a declared structure is at definition time. Here isan example using the preceding “address” structure.

```
struct address
{
        char street[20];
        char city[20];
        char state[10];
        int zip;
}homeaddr;
```

The preceding declaration of “homeaddr” causes storage to be allocated to hold a structure matching the data area required to hold each field in the “struct address” structure definition.

Another way to have caused a new structure to be allocated is to have made the following definition within the source code after the “address” structure had been defined:

```
struct address workaddr;
```

Although not to be discussed in detail here, you can also declare an array of structures in “C”. Here’s an example:

```
struct address alladdr[25];
```

The declaration above declares a variable of 25 structures having the type of “struct address”.

Assigning Structures
In “C”, you assign values to a structure when you define it or anytime within the body of your source code. If you do it a definition time, you must use a similar method discussed with initializing arrays. Here’s an example:

```
struct address workaddr = {“Main Street”, “Los Angeles",
“CA”, 90211};
```

Within the source program, you can only make structure assignments by assigning a structure to another structure or by assigning each individual member of the structure to their own values. Here’s an example of doing both:

```
workaddr2 = workaddr;

workaddr3.street = “Beach Blvd”;
workaddr3.city = “Miami”;
workaddr3.state = “FL”;
workaddr3.zip = homeaddr.zip;
```

Note that the use of the period “.” is required to access individual members of a structure within the source code (as shown in the preceding example.

Pointer to Structures
In “C”, a pointer to a structure can be declared much the same way as pointers to other data types. Here is an example:

```
struct address myaddr, *addressptr=&myaddr;
```

In the preceding example, “addressptr” is a pointer of type “struct address” which is also assigned to the same storage location as “myaddr”. Although an assignment was made, it did not have to be made at the time of the declaration.

Generally, in “C” as in SabreTalk, pointer structure variables should only be used in operations involving structures of the same declaration. Although not discussed here, an array of pointers to structures is also allowed in “C”.
Accessing the individual members of a pointer to a structure is different than with normal variables defined to a structure. In the case of pointer to structures, you must use the “->” operator to reference each item. Here’s another example using the pointer defined above.

```
addressptr->street = “Beach Blvd”;
addressptr->city = “Miami”;
addressptr->state = “FL”;
addressptr->zip = homeaddr.zip;
```

That about wraps up our review of data concepts. In the next and final installment, we’ll look at the coding an control statements you’ll need to be familiar with as you transition from SabreTalk to “C”.