Common C Pitfalls

We often encounter programmers who are new to C and call us with C-related programming problems. Here are a few of the many things to watch out for when programming with C.

• Code written in C can be simultaneously very powerful and very obscure. We have tried to write the C example programs in such a way that they maintain a line-by-line correspondence to the Fortran and Pascal example programs. If you don't understand what we are doing in C, try looking at the corresponding Fortran or Pascal source code.

• A missing ";" at the end of a line, or the use of "=" instead of "==" in a logical expression are common mistakes.

• C always uses 0-based array indexing, although array declarations are 1-based. For example, an array "X" of 10 integers would be dimensioned as "X[10]" but the 10 elements would be indexed from 0 to 9. Many C programmers find this counter-intuitive and add an extra element to each array so that they can use 1-based indexing!

• If a "case" in a "switch" block does not end in "break", then the following case is also executed. Most programmers find this to be counter-intuitive (the point of a case block is almost always to execute one case or another).

• The following expressions are equivalent, and both forms are used by C programmers: "(*gWindptr[0]).txFont" and "gWindptr[0]->txFont". Note, in particular, the need in this case to use parentheses with "*", since "." takes precedence over "*".

• Although the address of an array obtained with the "&" operator and the array name can be used interchangeably in C, we have, for consistency across compilers, avoided use of the "&" operator before array names. Note, however, that this equivalence of an array's address and the array name does not extend to "struct"s. For example, the toolbox type "Rect" is a struct, not an array, so a variable of this type should always be preceeded by "&" when passed as an argument in a toolbox call. Finally, if referencing the address of a nonarray element within an array, then the "&" operator must be used. For example, the toolbox call BlockMove expects a source address, destination address, and a number of bytes to move. If "a" and "b" are one-dimensional integer arrays, then writing

```
  BlockMove(&a[i], &b[j], 4L);
```

would move 4 bytes from the "a" array at element "i" to the "b" array at element "j". Without the "&" operator,

```
  BlockMove(a[i], b[j], 4L);
```

would move 4 bytes from the address given by "a[i]" to the address given by "b[j]" (probably not what you intended!). On the other hand, if "a" and "b" were arrays of strings (arrays of character arrays), then writing

```
  BlockMove(&a[i], &b[j], 4L);
```

would produce the same result as

```
  BlockMove(a[i], b[j], 4L);
```

This can be quite confusing.

• When reading and writing integers with "printf" and "scanf" (or related functions), "long" (4-byte) integers must be assigned an "l" (lower case L) as the size specification in the corresponding conversion specification of the format string.

• If not using prototypes, you must be careful to pass the proper integer type as arguments in function calls. For example, the FaceIt dispatching procedure expects "long" (4-byte) arguments, so you should not pass it "short" (2-byte) integers. Be careful, in particular, when passing constants since these default to short integers unless followed by "L":

```
   FaceIt(0,DoLoop,0,0,0,0);                /* WRONG!!! */
   FaceIt(0L,DoLoop,0L,0L,0L,0L);          /* RIGHT!!! */
```