TransSkel Programmer's Notes

4: TransSkel and Function Prototypes

Who to blame: Paul DuBois, dubois@primate.wisc.edu

Note creation date: 02/16/93
Note revision: 1.02
Last revision date: 01/06/94
TransSkel release: 3.02

The TransSkel header and source files use function prototypes to provide the compiler with valuable information. It's best that you make use of this information by including *TransSkel.h* when compiling any of your own source files in which TransSkel functions are used. If you don't, you may end up with programs that compile properly but likely don't run correctly — and fail in subtle and hard to catch ways. This Note explains why.

C and prototypes

In C, you have a choice of two ways of defining and declaring functions. You can use the syntax of Kernighan and Ritchie's original description of C, or the newer ANSI C syntax. The K&R and ANSI styles are referred to here as the old and new styles, respectively.

C functions, as originally described by Kernighan and Ritchie, are defined like this:

```
functype f (arg1, arg2)
argtype1     arg1;
argtype2     arg2;
{
       ...
}
```

and declared (e.g., in a header file) like this:

```
functype f ();
```

`functype` is the return value type of `f` and `argtype1` and `argtype2` are the types of the arguments. (The number of arguments may vary, of course.) If `functype` is missing from the definition or declaration, `f` is assumed to return `int`. A function is also assumed to return `int` if it's used before a definition or declaration for it has been seen.

Although the declaration can indicate the function's return value type, it doesn't say anything about the argument types, or even whether there are any.

Using ANSI C, the same function can be defined like this:

```
functype f (argtype1 arg1, argtype2 arg2)
```

```
{
        . . .
}
```

and is declared using a prototype like this:

```
        functype f (argtype1 arg1, argtype2 arg2);
```

Prototypes affect compiler behavior in two important ways: (i) the compiler can perform more complete type checking on function calls, since it can check not only the function return value type, but also the number and types of the arguments; (ii) the compiler is free to optimize the way it passes arguments.

When K&R C declarations are used, the compiler performs promotion or "widening" of arguments passed to function calls. For instance, `char` and `short` are widened to `int`, `float` is widened to `double`. If the compiler is provided with ANSI function prototypes instead, it is free to pass arguments in the most efficient way. In particular, it may forego the argument widening that occurs in K&R C. A `char`, for instance, is widened to `int` when passed to a function for which there is no prototype, but if there is a prototype available indicating an argument is a `char`, it may well be passed as `char`, without promotion.

Problems can arise when the old and new styles are mixed, e,g., if a function is compiled to expect its arguments non-widened and calls to the function are compiled with widened arguments, or vice-versa. Suppose a function takes a `char` argument and is defined using an ANSI style definition. It will expect to see an `char`-width argument. If calls to the function are compiled without prototype information, widening will occur and an `int`-width argument will be passed. The result is that the program may compile and link properly, but fail to run correctly. Problems of this sort can be quite difficult to track down.

It's best to use prototyped declarations together with ANSI style definitions, or else unprototyped declarations together with K&R style definitions to avoid mixing old and new styles and getting bitten as described above. The discussion below describes how this general principle applies to working with TransSkel.

Prototypes in TransSkel

TransSkel is written and compiled using ANSI function definitions and prototyped declarations. You should use the prototype information when compiling any of your own source containing calls to TransSkel routines. Failure to do so can result in mismatches of the sort described above.
There are two things to watch out for:

- Your application should call TransSkel functions in a way that's consistent with the way TransSkel expects them to be called. `SkelDlogFilter()` is an example. Its prototype looks like this:

  ```
  ModalFilterProcPtr
  SkelDlogFilter (ModalFilterProcPtr filter, Boolean doReturn);
  ```

  `Boolean` in THINK C is a character (single byte) type. If you compile a call to this function without prototype information, the `Boolean` argument will be widened, but TransSkel is compiled not to expect widened parameters. The result of the mismatch is that `doReturn` always appears to be false to SkelDlogFilter(). (The symptom is that the return key doesn't select the default button in the dialog you're using `SkelDlogFilter()` with.)

- Functions in your application called by TransSkel should be compiled in a way consistent with the way TransSkel expects to call them. For instance, the window handler callback function that handles activate events is supposed to take a single argument, a `Boolean` indicating whether the window is coming active or not. The prototype for activate handlers looks like this:

```
    void (*SkelWindActivateProcPtr) (Boolean active);
```

Here too, since TransSkel is compiled using prototypes, the compiler is free to pass a `Boolean` as a single byte, without promotion. This means your activate functions should expect non-widened arguments. In order to avoid mixing old and new function styles, you should define your activate function this way:

```
    void Activate (Boolean active)
    {
            ...
    }
```

If you define it as given below instead, you’re still using a K&R-style function definition:

```
    void Activate (active)
    Boolean      active;
    {
            ...
    }
```

In this case, your routine will be compiled to expect a widened argument, i.e., it will expect a `Boolean` to be passed as an `int` and `active` will always appear to be `false`.

If you include *TransSkel.h* in your source files that call TransSkel functions, you’ll avoid function style inconsistencies, or at least you get a warning from the compiler when changes need to be made in your code. Your calls to TransSkel functions will be compiled using prototype information, and will be consistent with the way TransSkel is compiled. In addition, the compiler will know whether the functions you pass to TransSkel routines are defined the way TransSkel expects to call them. This is because it will know the prototypes for functions passed as arguments to TransSkel routines (e.g., the activate handler passed to the `SkelWindow()` function) and can compare them with the way your functions are actually defined to see if they are consistent. If they’re not, you will get a compilation error.

Conclusion: it’s pretty important that you use the information in *TransSkel.h*.

In some sense the requirement for using prototypes and ANSI style function definitions turns TransSkel from a benevolent servant into a harsh taskmaster. This is somewhat unfortunate. On the other hand, it’s not clear what a better solution might be. If TransSkel were written *without* prototypes, it would mean the programmer would need to *not* use prototypes for TransSkel-related routines, such as window and menu handler callback functions. So no matter which way TransSkel were written, it would enforce a discipline on the use of its routines and callbacks to those routines.