

MATH0129: EXERCISE 8

Vassilis Kostakos

12 April 2000

Short summary

My report has the following parts:

1. Differences between K&R C and ANSI C
2. Making the conversion (tools, methods, etc)
3. Advantages and disadvantages of converting from K&R C to ANSI C
4. Conclusion

1 Differences between K&R C and ANSI C

Differences between ANSI and K&R fall into four categories:

- Prototype change – parameters must be written in the function declaration, not just the definition.
- Keyword changes – several new keywords were added including enum, const, volatile, signed, and void.
- Quiet changes – which are mostly minor changes which cause code to still compile but perhaps operate in a different manner.
- Everything else – these changes will have almost no effect and will almost never be encountered in practice (eg. trigraphs can be used to represent a single character by using three characters).

A variety of minimum sizes were defined by the ANSI standard including:

- 31 parameters in a function definition
- 31 arguments in a function call
- 509 characters in a source line
- 32 levels of nested parentheses in an expression
- long ints are at least 32 bits

2 Making the coversion

In this section I will try to point out some methods and solutions to overcome the problems of converting K&R C to ANSI C.

There exist some utilities which will assist in the production of function prototype, and analysing the original code. A program called CPROTO was posted to comp.sources.misc in March, 1992. There is another similar program called CEXTRACT. Many vendors supply simple utilities like these with their compilers. However, one must be careful when generating prototypes for old functions with “narrow” parameters.

The ANSI C compiler allows both old-style and new-style C code. The following -X (note case) options provide varying degrees of compliance to the ANSI C standard.

- Xa** (a = ANSI) ANSI C plus K&R C compatibility extensions, with semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler issues warnings about the conflict and uses the ANSI C interpretation. This is the default mode.
- Xc** (c = conformance) Maximally conformant ANSI C, without K&R C compatibility extensions. The compiler issues errors and warnings for programs that use non-ANSI C constructs.
- Xs** (s = K&R C) The compiled language includes all features compatible with pre-ANSI K&R C. The computer warns about all language constructs that have differing behavior between ANSI C and K&R C.
- Xt** (t = transition) ANSI C plus K&R C compatibility extensions, without semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler issues warnings about the conflict and uses the K&R C interpretation.

For an existing application to benefit from function prototypes, there are a number of possibilities for updating, depending on how much of the code you would like to change:

1. Recompile without making any changes. Even with no coding changes, the compiler warns you about mismatches in parameter type and number when invoked with the -v option.
2. Add function prototypes just to the headers. All calls to global functions are covered.
3. Add function prototypes to the headers and start each source file with function prototypes for its local (static) functions. All calls to functions are covered, but doing this requires typing the interface for each local function twice in the source file.

4. Change all function declarations and definitions to use function prototypes. For most programmers, choices 2 and 3 are probably the best cost/benefit compromise. Unfortunately, these options are precisely the ones that require detailed knowledge of the rules for mixing old and new styles.

2.1 Mixing Considerations

For function prototype declarations to work with old-style function definitions, both must specify functionally identical interfaces or have compatible types using ANSI C's terminology.

For functions with varying arguments, there can be no mixing of ANSI C's ellipsis notation and the old-style `varargs()` function definition. For functions with a fixed number of parameters, the situation is fairly straightforward: just specify the types of the parameters as they were passed in previous implementations.

In K&R C, each argument was converted just before it was passed to the called function according to the default argument promotions. These promotions specified that all integral types narrower than `int` were promoted to `int` size, and any float argument was promoted to double, hence simplifying both the compiler and libraries. Function prototypes are more expressive—the specified parameter type is what is passed to the function.

Thus, if a function prototype is written for an existing (old-style) function definition, there should be no parameters in the function prototype with any of the following types: `char`, signed `char`, unsigned `char`, `float`, `short`, signed `short`, unsigned `short`

There still remain two complications with writing prototypes: *typedef* names and the promotion rules for narrow unsigned types. If parameters in old-style functions were declared using *typedef* names, such as `off_t` and `ino_t`, it is important to know whether or not the *typedef* name designates a type that is affected by the default argument promotions. For these two, `off_t` is a long, so it is appropriate to use in a function prototype; `ino_t` used to be an unsigned short, so if it were used in a prototype, the compiler issues a diagnostic because the old-style definition and the prototype specify different and incompatible interfaces.

Just what should be used instead of an unsigned short leads us into the final complication. The one biggest incompatibility between K&R C and the ANSI C compiler is the promotion rule for the widening of unsigned `char` and unsigned `short` to an `int` value. The parameter type that matches such an old-style parameter depends on the compilation mode used when you compile:

-Xs and -Xt should use unsigned `int`

-Xa and -Xc should use `int`

The best approach is to change the old-style definition to specify either `int` or unsigned `int` and use the matching type in the function prototype. You can always assign its value to a local variable with the narrower type, if necessary, after you enter the function.

2.2 Unsigned versus value preserving

According to K&R, *The C Programming Language* (First Edition), unsigned specified exactly one type; there were no unsigned chars, unsigned shorts, or unsigned longs, but most C compilers added these very soon thereafter. Some compilers did not implement unsigned long but included the other two. Naturally, implementations chose different rules for type promotions when these new types mixed with others in expressions.

In most C compilers, the simpler rule, "unsigned preserving," is used: when an unsigned type needs to be widened, it is widened to an unsigned type; when an unsigned type mixes with a signed type, the result is an unsigned type. The other rule, specified by ANSI C, is known as "value preserving," in which the result type depends on the relative sizes of the operand types. When an unsigned char or unsigned short is widened, the result type is int if an int is large enough to represent all the values of the smaller type. Otherwise, the result type is unsigned int. The value preserving rule produces the least surprise arithmetic result for most expressions.

2.3 Standard Headers and Reserved Names

Early in the standardization process, the ANSI Standards Committee chose to include library functions, macros, and header files as part of ANSI C. While this decision was necessary for the writing of truly portable C programs, a side effect is the basis of some of the most negative comments about ANSI C from the public—a large set of reserved names. This section presents the various categories of reserved names and some rationale for their reservations. At the end is a set of rules to follow that can steer your programs clear of any reserved names.

To match existing implementations, the ANSI C committee chose names like `printf` and `NULL`. However, each such name reduced the set of names available for free use in C programs. On the other hand, before standardization, implementors felt free to add both new keywords to their compilers and names to headers. No program could be guaranteed to compile from one release to another, let alone port from one vendor's implementation to another. As a result, the Committee made a hard decision: to restrict all conforming implementations from including any extra names, except those with certain forms. It is this decision that causes most C compilation systems to be almost conforming. Nevertheless, the Standard contains 32 keywords and almost 250 names in its headers, none of which necessarily follow any particular naming pattern.

3 Advantages and Disadvantages

ANSI C's most sweeping change to the language is the function prototype borrowed from the C++ language. By specifying for each function the number and types of its parameters, not only does every regular compile get the benefits of argument and parameter checks (similar to those of lint) for each function call,

but arguments are automatically converted (just as with an assignment) to the type expected by the function. ANSI C includes rules that govern the mixing of old- and new-style function declarations since there are many, many lines of existing C code that could and should be converted to use prototypes. However, the fact that an existent application is written using K&R C, means that a lot of effort has to be given so that the conversion can happen. This implies expenses for the company, which could turn out to be worthless. Furthermore, it could be the case that as a result of the conversion, the program will behave slightly different.

4 Conclusion

As it was pointed out, there are various advantages and disadvantages regarding the conversion of a large program from K&R C to ANSI C. It all comes down to weighting the two, and deciding which is more desirable. The company could spend resources on translating the existing program, instead of writing new programs. Of course, the label of ANSI C on a program is always an advantage, since it implies portability and familiarity of code. Of course, it could be the case that the conversion is not needed at all. But this depends on the company's plans for the future, and how and where the company intends to use the product.