Generic Packages in C

Dick Grune

Vrije Universiteit
de Boelelaan 1081
1081 HV Amsterdam

ABSTRACT

The structuring achieved by generic packages in Ada can be cheaply emulated in C by judicious use of the preprocessor. Two files are required for the generic package: the specification and the body. Two more files are used in the instantiation: one holding the instantiation parameters and one with auxiliary code. Instantiation results in normal C header and object files (`*.h` and `*.o`). Dependency control can be delegated to the *make* progam.

Keywords: UNIX, preprocessors, Ada.

Packages, Instantiations and Recompilation.

The designers of Ada* have recognized that it would be useful to allow types as parameters. The standard example is that of a sorting routine, parametrized with the type of the objects to be sorted, and a comparison routine between them. The sorting algorithm could then be varied independently of the kind of objects, thus reducing the complexity of the software needed.

Using types as parameters is a facility that is provided by very few languages, and by no compiling language I know of. Its implementation is not impossible, but is awkward; a formal type parameter could be implemented as a vector of a number of routine thunks (see, e.g., Aho, Sethi and Ullman [1986]), for allocation, for several operators, etc. And unless one introduces types of types, static checking is jeopardized. For an extensive review of the problems and solutions see Bray [1983].

An alternative implementation is by brute force: isolate all calls to routines with type parameters and recompile their bodies for all pertinent types. (This is possible only if the types in any given program form a finite set and all calls concerned are spotted easily enough.) This forces us to view procedure calling as a two-stage process: first the appropriate procedure is instantiated from a generic package while incorporating the "heavy-weight" parameters, and then the resulting simple procedure is called with its "light-weight" parameters. Since Ada provides different syntax for the two stages, instantiation by recompilation is a possibility in Ada, and it is this approach that we can easily imitate in C, using the C preprocessor and the UNIX† utility *make* (Feldman [1979]).

Immediately the question is asked: "what are heavy-weight parameters" and a reasonable answer is: "those that affect syntactic correctness". Indeed the heavy-weight parameters in Ada are types, which allow or disallow certain operators as +, selection or *succ()*; and subprograms, which require a specific number of parameters. In our imitation we have the C compiler do the consistency checking for us.

It may be noted in passing that selectors cannot be generic parameters in Ada, although this is occasionally useful: a record may have two similar components on which similar actions are to be defined. Our technique has no problems in allowing this.

A Toy Problem

To demonstrate our technique, we shall use a package that keeps a top-10 list of (comparable) objects of as yet unknown type. The number of objects it is supposed to keep (`TOP_SIZE`), the object type (`TOP_TYPE`) and the name of the comparison function (`TOP_BETTER`) are its instantiation parameters. The package supplies the user with an initialization procedure, to be called when the top-10 competition starts, an insertion

* Ada is a Registered Trademark of the U.S. Government – Ada Joint Program Office.
† UNIX is a Registered Trademark of AT&T Bell Laboratories.

procedure which accepts an object and puts it on the list in the right place or rejects it, and an iterator which offers to the user the objects in best-to-worst order. The iterator has an open-procedure to start it, a next-function which produces the next object, and a close-procedure; the type of the iterator (which is private) and a constant `NoObject`, returned by the next-function at end-of-list are provided by the package. For iterators, see the CLU Reference Manual by Liskov et al. [1981].

The specification part of this package could be as follows (in pseudo-Ada, that is. Real Ada would not use `access TOP_TYPE` at all, and the iterator would be a package, thus doing away with the `OpenTopIter` and `CloseTopIter` routines. The form below is chosen to convey the flavour of the generic approach while still holding on to C-like data access.)

```
-- This is a generic package for keeping a top-10 list of objects of
-- some formal type.
-- Specification part:

generic                                     -- 3 formals
        TOP_SIZE: integer;                  -- the size of the top-N list
        type TOP_TYPE is private;           -- the type of the objects
        with function TOP_BETTER(i, j: access TOP_TYPE) return BOOLEAN;
                                            -- TRUE if object at i is better
                                            -- than object at j, FALSE otherwise
package TOP_PKG is

        procedure InitTop;                  -- clears the list

        procedure InsertTop(obj: access TOP_TYPE);
                                            -- accepts a pointer to an object

        -- an iterator to yield the objects in best-to-worst order:
        type TopIter is private;            -- to declare the iterator

        procedure OpenTopIter(ti: access TopIter);
                                            -- starts the iterator

        function NextTopObject(ti: access TopIter) return TOP_TYPE;
                                            -- yields next object and
                                            -- moves iterator

        NoObject: constant access TOP_TYPE:= NULL;
                                            -- to be yielded at end-of-list

        procedure CloseTopIter(ti: access TopIter);
                                            -- stops the iterator
private
        type TopIter is integer;            -- a simple counter

end TOP_PKG;
```

Such a package is useful for a variety of purposes like displaying the names of the 8 most recently modified files in a directory, keeping a list of "best" solutions in a heuristic search, etc. We shall apply it here in an absurdly simple example.

Suppose we have a rope of length `ROPE` with which we want to surround a rectangle of roughly maximum area. Our programmer, badly underpaid in this age of budget cuts, who unfortunately knows nothing about calculus, symmetry arguments or floating point numbers but who does understand generic packages, decides to write a simple loop generating all possible rectangles with circumference `ROPE` and offering each one to a top-3/compare-by-area version of the top-10 package. The top 3 are then printed by using the iterator. This example uses most of the features used in real applications.

The Generic Package in C

We have to distinguish three parties in this venture: the *author* of the generic package, the *instantiator* of the generic package to the normal package and the final *user* of the package. The final user is supposed to be an ordinary C programmer who is not interested in genericity and just wants to #include a header file and link in an object file.

For the generic package *X_pkg*, the author provides two files, *X_pkg.spec* and *X_pkg.body*, and a simple paradigm for the *Makefile*.

X_pkg.spec contains the specification in comment form, interlaced with the equivalents in C. *X_pkg.body* contains the body of the package, almost entirely in C, since code is its main contribution. Both files freely use the names of the instantiation parameters `TOP_SIZE`, `TOP_TYPE` and `TOP_BETTER`, knowing that their values will be available by the time the text reaches the C compiler. See the files *top_pkg.spec* and *top_pkg.body*.

The instantiator provides two files, *X.inst* and *X.code*. *X.inst* contains #defines for the instantiation parameters and *X.code* contains code and/or #includes specific to this instantiation.

Note that the prefix here is *X*, not *X_pkg*, since the two package names in the (pseudo-)Ada instantiation also differ:

```
package TOP is
        new TOP_PKG(
                TOP_SIZE => 3,
                TOP_TYPE => struct rectangle,
                TOP_BETTER => larger
        );
```

The actual instantiation is now very simple: *X.inst* and *X_pkg.spec* together contain all the information the final user will need (and slightly more): their concatenation forms the header file *X.h*. The files *X_pkg.spec*, *X_pkg.body*, *X.inst* and *X.code* compiled together will yield the object module *X.o* to be picked up by the loader. C declaration requirements virtually dictate the concatenation order *X.code*, *X.inst*, *X_pkg.spec*, *X_pkg.code*.

The final user can now #include *X.h* in his program to get access to the names in the specification part. When applied, these names may be replaced by the form they have taken on after the instantiation.

Example: in *rectangle.c*, the user compares a result to something called `NoObject`:

```
while (ra != NoObject)
```

Now `NoObject` has been #defined in *top_pkg.spec* as `((TOP_TYPE *) 0)`, i.e., a nil pointer of type `TOP_TYPE`, which in turn was #defined as `struct rectangle` in *top.inst*. Consequently the actual comparison will read:

```
while (ra != ((struct rectangle *) 0))
```

which is exactly right.

The Makefile

The Ada Programming Support Environment (APSE) is envisaged with a mechanism for keeping object modules up to date when specifications change. In our setup this task can easily be performed by the UNIX program *make*. Very summarily sketched, a *Makefile* is a list of statements of the form "File *A* should be newer than files *B*, *C*, ...", augmented by commands to execute when *A* is older than *B* or *C*; these commands should have the effect of making the statement true.

A call of *make* will examine the *Makefile* and perform the necessary update commands.

An entry like

```
top.h:  top.inst top_pkg.spec
        cat top.inst top_pkg.spec >top.h
```

will keep *top.h* up to date in the face of changes in *top.inst* (the instantiator changed some instantiation parameters) or in *top_pkg.spec*.

For more details see the example.

Limitations

The C preprocessor is essentially a simple macro processor featuring a flat name space: a name that exists is visible everywhere. It is clearly impossible to follow Ada scope rules this way, but if the user uses unique names only, all is well.

A special problem arises if we want to instantiate two or more packages from one generic package: the generated names have to differ. In Ada the proper name is made by qualifying the generic name:

```
Money.InsertTop(40.13)
```

vs.

```
Name.InsertTop("Aardvark")
```

in a context where there are top-10 lists both for salaries and for names.

Here we need slightly more help from the C preprocessor than we are entitled to: many C preprocessors allow us to construct identifiers. We then view the name of the instantiated package as one of the parameters of the instantiation:

```
#define TOP_NAME()      Money
```

and qualify the public names in *top_pkg.** thus:

```
TOP_NAME()_InsertTop(obj) register TOP_TYPE *obj;       {
```

Hopefully the C preprocessor will oblige and produce:

```
Money_InsertTop(obj) register struct money *obj;        {
```

This does not solve the problem of using both instantiated packages in the same file: identifiers defined by `#define` in the two generated header files *Money_top.h* and *Name_top.h* may collide. More powerful instantiation can be done with the UNIX stream editor *sed*.

It will be clear that some checking will be left undone. For instance, inside *top_pkg.body* we could use the selector `height` from *rectangle.h*, and in this instantiation neither the C compiler nor *lint* (a more powerful context-sensitive checker) would find anything amiss. Such usage would, however, look out of place in *top_pkg.body*, which is solely concerned with administration issues, and it would be caught by the compiler in any different instantiation.

Efficiency

Since no trace of the above trickery shows up in the object code, the run-time speed of the resulting program will be the same as that of a program constructed by traditional means. The size of the object program might be considered a problem, since if the package is instantiated *N* times in the same program, the object will contain *N* almost identical copies. The situation, however, is probably not worse than in Ada, where one would also expect each instantiation to yield a separate object module. (In Ada one could do better and save code by really passing subprograms as parameters, but that would then adversely affect the run-time speed.) Moreover, normally *N* will be quite small.

Likewise, the compiler speed situation is not worse than in Ada (and since C compilers are much faster, it is actually a lot better!) The time taken by the C preprocessor is negligible, but for each instantiation the result has to be compiled. Proper use of the *Makefile* will, however, ensure that only the necessary recompilations will take place.

And finally, the disk space occupied by the several files produced by each instantiation might be cause for worry, but here we can do something: rather than concatenate some files *A*, *B* and *C*, we can produce a short file with the text

```
#include        "A"
#include        "B"
#include        "C"
```

which will serve just as well.

Comparison to other systems

Boyd [1984] and Stroustrup [1982] provide generics in a form similar to Simula classes. Boyd's environment is that of *Modular C*, a form of C "revamped" to look like Ada in some respects, by using the C preprocessor. A distinction is made between "free" generics, which works through dynamic instantiation, and "bound" generics, a variant intended for static instantiation. Our technique is more modest and directly applicable in that it works in unadorned C but does not allow dynamic instantiation (nor does Ada for that matter, except by using tasks.)

Stroustrup's system comprises a special-purpose preprocessor which extends C with "classes" a la Simula, thus providing packages of data structures with access routines. Again instantiation of classes can be dynamic. His technique for generic classes is essentially the same as ours.

Dutta [1985] is concerned with uniqueness of names, when duplicating functions. Privacy is acquired by putting the addresses of functions in a struct: this avoids name clashes by effectively creating a two-level name space. A function `func` in a module `keypad` is called `(*keypad.func)()`, much in keeping with the Ada notation `keypad.func()`. Since only the normal C preprocessor is used, our technique can coexist with this approach.

Acknowledgements

I thank Erik Baalbergen, Henri Bal, Ceriel Jacobs and Andy Tanenbaum for commenting on drafts of this paper.

Literature

Aho, Sethi and Ullman [1986]
Aho, A.V., Sethi, R. and Ullman, J.D., "Compilers – Principles, Techniques and Tools", Addison-Wesley Publ. Comp., 1986.

Boyd [1984]
Boyd, S, "Free & bound generics, two techniques for abstract data types in Modular C", SIGPLAN Notices, 19, #3, March 1984.

Bray [1983]
Bray, G., "Implementation Implications of Ada Generics", Ada LETTERS, 3, #2, Sept/Oct 1983.

Dutta [1985]
Dutta, K., "Modular programming in C: an approach and an example", SIGPLAN Notices, 20, #3, March 1985.

Feldman [1979]
Feldman, S.I., "Make – A Program for Maintaining Computer Programs", Software – Practice & Experience, 9, #4, pp 255-266, April 1979.

Liskov et al. [1981]
Liskov, B.H. et al., "CLU Reference Manual", Lecture Notes Computer Science 114, Springer Verlag, 1981.

Stroustrup [1982]
Stroustrup, B., "Classes, an Abstract Data Type Facility for the C Language", SIGPLAN Notices, 17, #1, Jan 1982.