C with Macros

James K. Lowden

ABSTRACT

We present a macro-processor proof-of-concept prototype, “macC”, for C. Unlike the standard C preprocessor, macC has access to the program’s abstract syntax tree. Consequently, a macro can evaluate its arguments to generate C functions. On completion, macC produces the modified C source code for compilation.

If implemented as part of a C compiler (instead of as a free-standing macro processor), such macros could solve long-standing irksome difficulties in C programming. For example, row-retrieval from an SQL DBMS can be completely automated. We include an example of such a macro.

27 March 2023

C with Macros

James K. Lowden

Introduction

So-called "macro processing" in C and C++ is today generally held in disfavor. The programmer is advised to use enumerated constants, inline functions, and templates wherever possible. But, to the author's knowledge, no C compiler has ever offered the programmer *true*, Lisp-like macros.

A C preprocessor macro — or C++ template, for that matter — has no access to the compiler's internal state. The preprocessor tokenizes the parameters, providing them as opaque strings to the macro body. The macro can neither interrogate the parameters, nor use their compiled definitions. It has only the strings.

In Lisp, the macro, on invocation, has access to the entire program state. A Lisp macro appears, syntactically, to be a function. But, whereas a function's arguments are processed before the function itself is invoked, a macro's arguments are provided to the macro verbatim, uncompiled. The macro determines how and whether the argument is interpreted. That control, combined with access to the entire program's compiled state, gives the Lisp macro power that is not available to the C programmer in any form.

A lesser form of the Lisp macro (one might say) is the Lisp function. Like a macro, a Lisp function has access to the compiled state of its arguments, and of the whole program. Functions in other languages, for example Python, share the same property.

The question arises: What could the C programmer do with macro processing that had access to the compiled state of the program? Terrible things, to be sure, because sharp implements are best used wisely. But also wonderful things.

Motivation

For the last 4 decades, database application programmers have been saddled with an unfortunate, tedious chore: fetching rows of data from the database into program memory. Barrels of ink, man-years of time, and not a few keyboards have been laid to waste in the attempt to facilitate that work. Much theorizing about so-called "impedance mismatch" of object-oriented languages and relational databases supposedly explained the problem. But no solution allowed the C programmer to write something like:

```
struct my_row_t my_row;
while( fetch_row(db) != NO_MORE_ROWS ) {
    row = db.row();
}
```

That assignment (of database data to program memory) cannot be implemented by any preprocessor or library because the fetched database data is an *array of runtime types and values*. The SQL sent to the DBMS determines both the number of columns in any result row, and the type of each column. The DBMS client library — ODBC, or other — converts the database datatype to a C datatype. There is no way for the DBMS client library synthesize a **`my_row_t`** object for the compiler to copy.

Typical of this drudgery is the ODBC **`SQLBindCol`** function:

```
SQLRETURN SQLBindCol(
     SQLHSTMT       statement_handle
     SQLUSMALLINT   column_number,
     SQLSMALLINT    c_data_type,
     SQLPOINTER     c_variable_address,
     SQLLEN         c_variable_size,
     SQLLEN *       database_data_size);
```

The programmer calls **`SQLBindCol`** once for each column in every query, laboriously explaining to the ODBC driver which buffer is to be filled, and the type of the target variable.† Although in any given situation, only one conversion is best (or even sensible), that determination is left to the programmer, because *the C language and the C preprocessor cannot interrogate the compiled definition of the assigned structure.*

Until now.

Technology

To modify a C compiler to implement macros as an extension to the C language is not an undertaking for the faint of heart. It's a mountain of work to provide an accessible API to the internals of the compiler for use by the macro author. And it's not a given that the work, if done, would be accepted or even welcomed by the compiler's developers and maintainers.

For that reason, macC is implemented as a preprocessor. We use the famous PLY parser by David Beasley, specifically the `pycparser` implementation.‡ The macC macro-processor first compiles the source code, taking note of defined structures and undefined functions. It imports the user's macros and, for each function prototype whose name matches that of an imported macro definition, it generates a function to replace the prototype in the parser's abstract syntax tree (AST). The processor then produces C source from the modified AST. Where there was once only a function declaration, there is now a definition.*

The astute reader will note the above description is not a macro in the Lisp sense. That is because, well, C is not Lisp.

The macC macros are more similar to a Lisp function. The `pycparser` has already parsed the code. The macro arguments are not provided verbatim; rather they are provided in their compiled state. That gives the macC macro author an opportunity to change the compiler's output, but not to change the order of evaluation.

Using the Macro

The example program couldn't be simpler:

† The example program is written for SQLite (https://www.sqlite.org/c3ref/funclist.html). Rather than a binding function, SQLite has a distinct function for every datatype conversion possible for a column. The only effect on the program is the name of the header file and the type representing the statement handle.

‡ https://www.dabeaz.com/ply/ and https://github.com/eliben/pycparser

* The `pycparser` has some limitations regarding the header files provided with the standard C library, which often use many nonstandard extensions to the C language. For that reason, the example `Makefile` removes the header files during parsing, and then re-introduces them on the command line for compilation.

```
struct sysrow_t {
  char type[16];
  char name[32];
  char tbl_name[32];
  int rootpage; };

int element_count( struct sysrow_t tgt );

void copy_row( struct sqlite3_stmt *stmt, struct sysrow_t *row );

/* ... */

  while( (erc = sqlite3_step(stmt)) == SQLITE_ROW ) {
    struct sysrow_t row;
    assert(element_count(row) <= sqlite3_column_count(stmt));
    copy_row(stmt, &row);
    print_row(&row);
  }
```

The program defines **`struct sysrow_t`** and has only prototypes for **`element_count`**, which returns how many members are in the passed structure, and **`copy_row`**, which copies the fetched data into the **`row`** variable. As an exercise, the reader may wish to consider how **`element_count`** might be conventionally implemented.

Normally, the programmer is required to define **`copy_row`**. It might look like this:

```
void copy_row( struct sqlite3_stmt *stmt, struct sysrow_t *row ) {
  int ordinal = 0;
  assert(4 <= sqlite3_column_count(stmt));

  strcpy( row->type, sqlite3_column_text(stmt, ordinal++) );
  strcpy( row->name, sqlite3_column_text(stmt, ordinal++) );
  strcpy( row->tbl_name, sqlite3_column_text(stmt, ordinal++) );
  row->rootpage = sqlite3_column_int64(stmt, ordinal++); }
```

The function ensures that the returned was produced by a query that created at least 4 columns, and copies them one by one into the **`row`** variable. This is the error-prone tedium we seek to avoid.

Implementing the Macro

Because macC is implemented in Python atop `pycparser`, the convenient macro implementation language is Python. Nothing *requires* Python; it's simply a kind of *lingua franca* for explicating the functionality.

Taking the simple example first, here is **`element_count`**:

```
def struct_member_count( function_name, meta, tgt ):
    struct = meta[ tgt[0] ]
    args = [ formal_val_decl( tgt[0], tgt[1] ) ]
    body = [ function_return( 'int', len(struct.elems) ) ]
    func = function_definition( function_name, args, body, 'int' )
    return func

def element_count( meta, tgt ):
    return struct_member_count( 'element_count', meta, tgt )
```

When a macC macro is invoked, it is always passed a dictionary of metadata describing all C `struct`s found in the original source file (technically, *translation unit*.) Each parameter is a tuple of `(type_name, parameter_name)`. **`element_count`** invokes the generalized “count-structure-elements” function, which begins by looking up the metadata for the type of its parameter in the metadata dictionary. With that information, it produces an array of formal parameters, and a function body. The body simply returns the number of members in the structure, in this case for **`struct sysrow_t`**. The name, parameters, body, and return type are passed to **`function_definition`**, which returns an AST node.

Finally, the macro author adds the macro’s name to a list of “exported” macros:

```
macros['element_count'] = element_count
```

When it imports the user’s macro definitions, macC relies on the imported module’s **`macros`** dictionary to denote which imported functions represent macro definitions.

Now onto the meat of the matter, the **`copy_row`** macro:

```
def copy_any_row( function_name, meta, stmt, tgt, return_type ):
    struct = meta[ tgt[0] ]
    nodes = []

    for i, elem in enumerate(struct.elems):
        node = None
        if elem.ctype == 'char' and elem.csize > 0:
            node = copy_string( tgt[1], elem.cname, stmt[1], i )
        elif elem.ctype == 'int':
            node = copy_int( tgt[1], elem.cname, stmt[1], i )
        else:
            raise Exception( "unrecognized type %s" % (elem.ctype)
)

        nodes.append(node)

    args = [ formal_ptr_decl( stmt[0], stmt[1] ),
             formal_ptr_decl( tgt[0], tgt[1] ) ]

    return function_definition( function_name, args, nodes,
return_type )

def copy_row( meta, stmt, tgt ):
    return copy_any_row( 'copy_row', meta, stmt, tgt, 'void' )
```

This is the same, only more. The macro calls the generalized “copy row to struct” function, which retrieves the metadata for its 2nd argument, which in this case is **`struct sysrow_t`**. It iterates over that structure’s elements. For each element, it calls a function that generates an AST node that copies the data. For example, **`copy_string`** produces an AST node that calls **`strcpy`**`(3)` using as a source the SQLite function that returns the column data as a C string.

As before, the function’s nodes are collected into a list, which is passed to **`function_definition`** to produce an AST representing the function. And, as before, the function is added to the export list:

```
macros['copy_row'] = copy_row
```

Invoking the Macro

The macC processor replaces the nodes that declare **`element_count`** and **`copy_row`** with nodes that defines them. On initialization, it makes a list of macro definitions, by name:

```
m = import_module(arg)
for name, mac in m.macros.items():
    macros[name] = mac
```

It then parses the C code into an AST, building a list of `struct` metadata. As it encounters function prototypes whose name matches its list of macros, it calls the macro definition of the same name. The macro definition returns an AST node representing a function definition. macC duly removed the function prototype and replaces it with a function definition.

When it's done, macC regenerates the C code. The macros produce:

```
int element_count(struct sysrow_t tgt)
{
  return 4;
}
```

and

```
void copy_row(struct sqlite3_stmt *stmt, struct sysrow_t *row)
{
  strcpy(row->type, sqlite3_column_text(stmt, 0));
  strcpy(row->name, sqlite3_column_text(stmt, 1));
  strcpy(row->tbl_name, sqlite3_column_text(stmt, 2));
  row->rootpage = sqlite3_column_int64(stmt, 3);
}
```

Future Work

The macros as presented are hardly more than a first draft. They have some limitations, none of which are inherent in the system, and which could be solved in various ways.

Perhaps the most obvious is that the **`copy_row`** macro is specific to a particular type. It also needs expanding to deal with other datatypes, and with SQL NULL.

Expanding the supported datatypes is relatively simple, merely mapping SQL datatypes to C datatypes. Support for SQL NULL is more a policy question: how does the structure represent NULL, since C has no data type that means *data is missing*. Because it's a macro, it could interrogate the target structure for a supported mechanism, perhaps a bitmap or (more conventionally) a Boolean associated with each member.

There is also a namespace issue: **`copy_row`** copies only one kind of row, **`struct sysrow_t`**, from one kind of DBMS, SQLite. The user is forced to invent a different name for each datatype. In the general case, the macro writer has to somehow come up with names for *copy a database row* for different DBMS APIs, too.

In C++, we have overloaded function names. In C, we have the `_Generic` preprocessor macro. The macro processor could, in principle, produce code for overloaded C functions under different names, perhaps using a variation of a C++ name mangler. It could also generate a matching`_Generic` preprocessor macro to allow simple, single-name use at the source-code level.

As discussed earlier, the most important limitation of macC is that it is not part of any C compiler. Integration with the compiler would presumably result in a system that executes faster, is easier to use, and is yet more powerful.

Conclusion

The ISO C standard lacks a macro processor that can interact with the compiler and compiled elements. It is limited to string manipulation. Which is to say, it is limited.

We have demonstrated both the utility and the feasibility of a "true" macro processor for C. Two use cases are presented, neither of which can be solved today without the programmer repeating things the compiler already knows, such as the number and type of the elements of a structure. C (and C++) programming could be made simpler and more reliable by introducing a macro processor, and standardizing a set of macros for common tasks that are otherwise impossible.