Example C++ User Type for Octave

June 2007

David Bateman

Table of Contents

1 User Types in Oct-Files

As Octave is written largely in C++, it can benefit from all of the object oriented programming capabilities of the C++ language. This makes it easy to add additional types to the Octave scripting language using the overloading capabilities of the C++ language.

Any of the existing Octave classes might be used as a basis for a new C++ class representing the new type. The basic class for the new user type to inherit from is `octave_base_value`, that contains all the default behaviors for the basic functionality of all Octave classes.

The following pages includes a simple example of a user type in an oct-file. It produces a new type for triangular matrices based on the `octave_matrix` class, and overloads certain operators including the left division operator, to allow rapid solution of linear equations including triangular

```
#include <octave/oct.h>
#include <octave/ov-re-mat.h>
#include <octave/ov-scalar.h>
#include <octave/ops.h>
#include <octave/ov-typeinfo.h>

static octave_base_value *
default_numeric_conversion_function (const octave_base_value& a);

class
octave_triangular_matrix : public octave_matrix
{
public:
  enum tri_type{
    Upper = 0,
    Lower = 1
  };

  octave_triangular_matrix (void) : octave_matrix(Matrix (0,0)) { };

  octave_triangular_matrix (const Matrix &m, tri_type t = Upper) :
    octave_matrix (m), tri (t)
  {
    octave_idx_type nr = m.rows ();
    octave_idx_type nc = m.cols ();
    if (tri == Upper)
      for (octave_idx_type j = 0; j < nc; j++)
        for (octave_idx_type i = j + 1; i < nr; i++)
          matrix (i, j) = 0;
    else
      for (octave_idx_type j = 0; j < nc; j++)
        for (octave_idx_type i = 0; i < j; i++)
          matrix (i, j) = 0;
```

```
}

octave_triangular_matrix (const octave_triangular_matrix& T) :
  octave_matrix (T), tri (T.tri) { };

~octave_triangular_matrix (void) { };

octave_base_value *clone (void) const
{
  return new octave_triangular_matrix (*this);
}

octave_base_value *empty_clone (void) const
{
  return new octave_triangular_matrix ();
}

octave_value subsref (const std::string &type,
                      const std::list<octave_value_list>& idx)
{
  octave_value retval;

  int skip = 1;

  switch (type[0])
    {
    case '(':
      retval = do_index_op (idx.front (), true);
      break;

    case '.':
      retval = dotref (idx.front ())(0);
      break;

    case '{':
      error ("%s cannot be indexed with %c",
             type_name().c_str(), type[0]);
      break;

    default:
      panic_impossible ();
    }

  if (! error_state)
    retval = retval.next_subsref (type, idx, skip);

  return retval;
```

```
}

octave_value_list dotref (const octave_value_list& idx)
{
  octave_value_list retval;

  std::string nm = idx(0).string_value ();

  if (nm == "type")
    if (isupper ())
      retval = octave_value ("Upper");
    else
      retval = octave_value ("Lower");
  else
    error ("%s can indexed with .%s",
           type_name().c_str(), nm.c_str());

  return retval;
}

// Need to define as map so tha "a.type = ..." can work
bool is_map (void) const { return true; }

octave_value subsasgn (const std::string& type,
                       const std::list<octave_value_list>& idx,
                       const octave_value& rhs)
{
  octave_value retval;

  switch (type[0])
    {
    case '(':
      {
        if (type.length () == 1)
          retval = numeric_assign (type, idx, rhs);
        else if (type.length () == 2)
          {
            std::list<octave_value_list>::const_iterator p =
              idx.begin ();
            octave_value_list key_idx = *++p;

            std::string key = key_idx(0).string_value ();

            if (key == "type")
              error ("use 'uppertri' or 'lowertri' to set type");
            else
```

```
                error ("%s can indexed with .%s",
                       type_name().c_str(), key.c_str());
            }
          else
            error ("in indexed assignment of %s, illegal assignment",
                   type_name().c_str ());
        }
        break;

      case '.':
        {
          octave_value_list key_idx = idx.front ();

          std::string key = key_idx(0).string_value ();

          if (key == "type")
            error ("use 'uppertri' or 'lowertri' to set matrix type");
          else
            error ("%s can indexed with .%s",
                   type_name().c_str(), key.c_str());
        }
        break;

      case '{':
        error ("%s cannot be indexed with %c",
               type_name().c_str (), type[0]);
        break;

      default:
        panic_impossible ();
      }

  return retval;
}

octave_base_value *try_narrowing_conversion (void)
{
  octave_base_value *retval = 0;

  if (matrix.nelem () == 1)
    retval = new octave_scalar (matrix (0, 0));
  else if (tri == Upper)
    {
      octave_idx_type nr = matrix.rows ();
      octave_idx_type nc = matrix.cols ();

      for (octave_idx_type j = 0; j < nc; j++)
```

```
        for (octave_idx_type i = j + 1; i < nr; i++)
          if (matrix.elem (i, j) != 0.0)
            {
              retval = new octave_matrix (matrix);
              break;
            }
    }
  else
    {
      octave_idx_type nc = matrix.cols ();

      for (octave_idx_type j = 0; j < nc; j++)
        for (octave_idx_type i = 0; i < j; i++)
          if (matrix.elem (i, j) != 0.0)
            {
              retval = new octave_matrix (matrix);
              break;
            }
    }
  return retval;
}

type_conv_fcn numeric_conversion_function (void) const
{
  return default_numeric_conversion_function;
}

bool isupper (void) const { return (tri == Upper); }
bool islower (void) const { return (tri == Lower); }
tri_type tritype (void) const { return tri; }

void assign (const octave_value_list& idx, const Matrix& rhs)
{
  octave_matrix::assign(idx, rhs);
}

octave_triangular_matrix transpose (void) const
  {
    return octave_triangular_matrix
      (this->matrix_value().transpose (),
       (tri == Upper ? Lower : Upper));
  }

void print (std::ostream& os, bool pr_as_read_syntax = false) const
  {
    octave_matrix::print (os, pr_as_read_syntax);
    os << (tri == Upper ? "Upper" : "Lower") << " Triangular";
```

```
      newline(os);
    }

private:
   tri_type tri;

   DECLARE_OCTAVE_ALLOCATOR
   DECLARE_OV_TYPEID_FUNCTIONS_AND_DATA
};

static octave_base_value *
default_numeric_conversion_function (const octave_base_value& a)
{
  CAST_CONV_ARG (const octave_triangular_matrix&);

  return new octave_matrix (v.array_value ());
}

DEFINE_OCTAVE_ALLOCATOR (octave_triangular_matrix);
DEFINE_OV_TYPEID_FUNCTIONS_AND_DATA (octave_triangular_matrix,
                                     "triangular matrix",
                                     "double");

DEFCONV (octave_triangular_conv, octave_triangular_matrix, matrix)
{
  CAST_CONV_ARG (const octave_triangular_matrix &);
  return new octave_matrix (v.matrix_value ());
}

DEFUNOP (uplus, triangular_matrix)
{
  CAST_UNOP_ARG (const octave_triangular_matrix&);
  std::cerr << "here\n";
  return new octave_triangular_matrix (v);
}

DEFUNOP (uminus, triangular_matrix)
{
  CAST_UNOP_ARG (const octave_triangular_matrix&);
  return new octave_triangular_matrix (- v.matrix_value (),
                                       v.tritype ());
}

DEFUNOP (transpose, triangular_matrix)
{
  CAST_UNOP_ARG (const octave_triangular_matrix&);
  return new octave_triangular_matrix (v.transpose ());
```

```
}

DEFBINOP(ldiv, triangular_matrix, matrix)
{
  CAST_BINOP_ARGS (const octave_triangular_matrix&,
                   const octave_matrix&);
  const Matrix m = v1.matrix_value ();
  const Matrix b = v2.matrix_value ();
  octave_idx_type nr = m.rows ();
  octave_idx_type nc = m.cols ();
  octave_idx_type b_nc = b.cols ();
  Matrix retval;

  if (nr != nc)
    error ("matrix must be square");
  else if (nr == 0 || nc == 0 || nr != b.rows ())
    error ("matrix dimension mismatch solution of linear equations");
  else
    {
      retval = b;
      if (v1.isupper ())
        {
          for (octave_idx_type j = 0; j < b_nc; j++)
            for (octave_idx_type k = nr - 1; k >= 0; k--)
              {
                if (retval (k, j) != 0.)
                  {
                    retval (k, j) /= m (k, k);
                    for (octave_idx_type i = 0; i < k; i++)
                      retval (i, j) -= retval (k, j) * m (i, k);
                  }
              }
        }
      else
        {
          for (octave_idx_type j = 0; j < b_nc; j++)
            for (octave_idx_type k = 0; k < nr; k++)
              {
                if (retval (k, j) != 0.)
                  {
                    retval (k, j) /= m (k, k);
                    for (octave_idx_type i = k + 1; i < nr; i++)
                      retval (i, j) -= retval (k, j) * m (i, k);
                  }
              }
        }
    }
```

```
  return retval;
}

DEFASSIGNOP (assign, triangular_matrix, triangular_matrix)
{
  CAST_BINOP_ARGS (octave_triangular_matrix &,
                   const octave_triangular_matrix&);
  v1.assign(idx, v2.matrix_value());
  return octave_value();
}

DEFASSIGNOP (assignm, triangular_matrix, matrix)
{
  CAST_BINOP_ARGS (octave_triangular_matrix &, const octave_matrix&);
  v1.assign(idx, v2.matrix_value());
  return octave_value();
}

DEFASSIGNOP (assigns, triangular_matrix, scalar)
{
  CAST_BINOP_ARGS (octave_triangular_matrix &, const octave_scalar&);
  v1.assign(idx, v2.matrix_value());
  return octave_value();
}

void
install_tri_ops (void)
{
  INSTALL_UNOP (op_uminus, octave_triangular_matrix, uminus);
  INSTALL_UNOP (op_uplus, octave_triangular_matrix, uplus);
  INSTALL_UNOP (op_transpose, octave_triangular_matrix, transpose);
  INSTALL_UNOP (op_hermitian, octave_triangular_matrix, transpose);

  INSTALL_BINOP (op_ldiv, octave_triangular_matrix,
                 octave_matrix, ldiv);

  INSTALL_ASSIGNOP (op_asn_eq, octave_triangular_matrix,
                    octave_matrix, assign);
  INSTALL_ASSIGNOP (op_asn_eq, octave_triangular_matrix,
                    octave_triangular_matrix, assignm);
  INSTALL_ASSIGNOP (op_asn_eq, octave_triangular_matrix,
                    octave_scalar, assigns);
}

static bool triangular_type_loaded = false;
```

```
static void
install_triangular (void)
{
  if (!triangular_type_loaded)
    {
      octave_triangular_matrix::register_type ();
      install_tri_ops ();
      triangular_type_loaded = true;
      mlock ("uppertri");
      mlock ("lowertri");
    }
}

// PKG_ADD: autoload ("istri", "uppertri.oct");
DEFUN_DLD (istri, args, ,
           "Returns true if argument is a triangular matrix")
{
  int nargin = args.length ();
  octave_value retval;

  if (nargin != 1)
    print_usage ();
  else if (! triangular_type_loaded)
    retval = false;
  else
    retval = args(0).type_id () ==
      octave_triangular_matrix::static_type_id ();
  return retval;
}

// PKG_ADD: autoload ("lowertri", "uppertri.oct");
DEFUN_DLD (lowertri, args, ,
           "Creates a lower triangular matrix")
{
  int nargin = args.length ();
  octave_value retval;

  if (nargin != 1)
    print_usage ();
  else
    {
      Matrix m = args(0).matrix_value();
      if (!error_state)
        {
          install_triangular ();
          retval = new octave_triangular_matrix
            (m, octave_triangular_matrix::Lower);
```

```
        }
    }
  return retval;
}

DEFUN_DLD (uppertri, args, ,
           "Creates an upper triangular matrix")
{
  int nargin = args.length ();
  octave_value retval;

  if (nargin != 1)
    print_usage ();
  else
    {
      Matrix m = args(0).matrix_value();
      if (!error_state)
        {
          install_triangular ();
          retval = new octave_triangular_matrix
            (m, octave_triangular_matrix::Upper);
        }
    }
  return retval;
}
```

It should be noted that the above class, is given as an example only, as Octave includes the ability to identify triangular matrices and use the appropriate method for the left division operator. It is however useful as an example.

Before the new user type can be used, it has to be registered as a valid Octave type. This is performed by the `register_type` method in the `install_triangular` function above. The `install_triangular` function also sets up the overloaded functions in the `install_tri_ops` function. A number of functions are overloaded, including the transpose, unary operators and the left division operators. Also note that there are three different assignment operators defined.

Finally, the constructor functions are locked in to Octave memory with the `mlock` function. The reason the constructor functions need to be locked into memory is that

```
a = uppertri (randn (3,3));
clear uppertri
disp(a)
```

would result in `a` being left defined, but the type it refers to being cleared from Octave's memory. The next reference to `a`, will then result in Octave exiting abnormally.

Note that in most ways user types have the same capabilities as any other Octave type. There are a few minor points were user types differ. Firstly, as the `octave_value` class has no knowledge of the user type, extracting the user type from the `octave_value` or setting the `octave_value` to the user type is necessarily different. Extracting the user type from the `octave_value` is done with

```
const octave_triangular_matrix ((const octave_triangular_matrix&)
        args(0).get_rep();
```

and setting the `octave_value` is done with

```
octave_value retval = new octave_triangular_matrix (...)
retval.maybe_mutate ();
```

It should be noted that the normal `octave_value` constructors call the `maybe_mutate` method, that check whether the previous operations should result in the type of the returned value being changed, using the `try_narrowing_conversion` function. The constructor functions in the example code, know the type won't change and so can ignore te `maybe_mutate` functions.

Two other important functions in the above class are the `subsref` and `subsasgn` functions. The `subsref` function is used for indexed referencing of variable of the user type such as `a (1)`, `a.type` or even `a (1).type`. The `subsasgn` function is similar, but is used for indexed assignments to the user type (ie. when the variable is on the left-hand side of the assigment).

It should be noted that to allow assignments such as

```
a = uppertri (...);
a.type = ...;
```

to work as expected, then the user type must be declared as a map by having the `is_map` method return true. The use of ".type" above to reference the type of the triangular matrix, is not necessarily recommended, but is rather an indication of how this type of indexing might be implemented.

A final important function included in the above class is the `numeric_conversion_function`. This returns a pointer to a function, that converts the triangular matrix to a normal Octave matrix. This allows the operators for normal Octave matrices to be applied to the user defined triangular matrices, with the return type being a normal Octave matrix type.