John Romero Programming Proverbs

-

- John Romero, “The Early Days of Id Software - John Romero @ WeAreDevelopers Conference 2017”

Creating shared libraries under GNU/Linux

- focus on the major advantage

- interfacing C, C++, Modula-2 with scripting languages
  - Python, Perl, Ruby, TCL
  - further focus examples around Python

Creating shared libraries under GNU/Linux

- Python's modules are either written in Python or are implemented as a shared library
  - or a combination of both

- we will briefly examine the following tools
  - `gcc`, `g++`, `libtool`, `swig`, `make` and `gm2`

Simple pedagogical example

- let us create a module to sum two integers, we will use `swig` to call C functions from Python

- **mymodule.i**

```
%module mymodule
%{
extern int sum (int a, int b);
%}
extern int sum (int a, int b);
```

Simple pedagogical example

- **mymodule.c**

```
int sum (int a, int b)
{
  return a + b;
}
```

Simple pedagogical example

- ```
  $ swig -python mymodule.i
  ```

- generates the following files:
  - `mymodule_wrap.c` and `mymodule.py`
  ```

Use gcc and libtool to compile and link the shared library

```
$ libtool --tag=CC --mode=compile gcc -g -I/usr/include/python2.7 \
  -c mymodule_wrap.c -o mymodule_wrap.lo
$ libtool --tag=CC --mode=compile gcc -g -I/usr/include/python2.7 \
  -c mymodule.c -o mymodule.lo
$ libtool --tag=CC --mode=link gcc -g mymodule.lo mymodule_wrap.lo \
  -rpath `pwd` -lc -lm -o libmymodule.la
$ cp .libs/libmymodule.so _mymodule.so
```

Use gcc and libtool to compile and link the shared library

- **testsum.py**

```
#!/usr/bin/python

import mymodule

print mymodule.sum (1, 2)
```

Use gcc and libtool to compile and link the shared library

```
$ python testsum.py
3
```

libtool on GNU/Linux

- notice the file extensions `.lo` and `.la`
- `libtool` is told about the library dependents and where other shared libraries reside

Output from running previous libtool command

```
libtool: compile:  gcc -g -I/usr/include/python2.7 -c mymodule_wrap.c  -fPIC -DPIC -o .libs/mymodule_wrap.o
libtool: compile:  gcc -g -I/usr/include/python2.7 -c mymodule_wrap.c -o mymodule_wrap.o >/dev/null 2>&1
libtool: compile:  gcc -g -I/usr/include/python2.7 -c mymodule.c  -fPIC -DPIC -o .libs/mymodule.o
libtool: compile:  gcc -g -I/usr/include/python2.7 -c mymodule.c -o mymodule.o >/dev/null 2>&1
libtool: link: rm -fr  .libs/libmymodule.a .libs/libmymodule.la .libs/libmymodule.lai \
  .libs/libmymodule.so .libs/libmymodule.so.0 .libs/libmymodule.so.0.0.0
libtool: link: gcc -shared  -fPIC -DPIC  .libs/mymodule.o .libs/mymodule_wrap.o \
   -L`pwd`/lib64 -lc -lm    -Wl,-soname -Wl,libmymodule.so.0 -o .libs/libmymodule.so.0.0.0
libtool: link: (cd ".libs" && rm -f "libmymodule.so.0" && ln -s "libmymodule.so.0.0.0" "libmymodule.so.0")
libtool: link: (cd ".libs" && rm -f "libmymodule.so" && ln -s "libmymodule.so.0.0.0" "libmymodule.so")
libtool: link: ar cru .libs/libmymodule.a  mymodule.o mymodule_wrap.o
libtool: link: ranlib .libs/libmymodule.a
libtool: link: ( cd ".libs" && rm -f "libmymodule.la" && ln -s "../libmymodule.la" "libmymodule.la" )
```

- note that this is the output from the slide containing the three `libtool` commands
- `libtool` is a highly portable mechanism to compile and link shared libraries
  - the output from the `libtool` commands will be different under OSX and Windows and or different versions of `gcc` and `g++`

John Romero Programming Proverbs

-

- John Romero, "The Early Days of Id Software - John Romero @ WeAreDevelopers Conference 2017"

PGE and libtool

- examine the file `pge/c/Makefile.am`
  - notice the rule starting with the text
  - `libpgeif.la:` this rule generates the library `libpgeif.la` using a variant of the command given on the previous slides

PGE and libtool

```
swig -outdir . -o pgeif_wrap.cxx -c++ -python $(top_srcdir)/i/pgeif.i
$(LIBTOOL) --tag=CC --mode=compile g++ -g -c pgeif_wrap.cxx \
  -I/usr/include/python$(PYTHON_VERSION) -o pgeif_wrap.lo
gm2 -c -g -I$(SRC_PATH_PIM) -fcpp -fmakelist \
  -I$(top_srcdir)/m2 $(top_srcdir)/m2/pgeif.mod
gm2 -c -g -I$(SRC_PATH_PIM) -fcpp -fmakeinit -fshared \
  -I$(top_srcdir) $(top_srcdir)/m2/pgeif.mod
$(LIBTOOL) --tag=CC $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) \
  --mode=compile gcc -c $(CFLAGS_FOR_TARGET) $(LIBCFLAGS) \
  $(libgm2_la_M2FLAGS) $(srcdir)/pgeif.c -o pgeif.lo
$(LIBTOOL) --tag=CC --mode=compile g++ -g -c _m2_pgeif.cpp -o _m2_pgeif.lo
$(LIBTOOL) --tag=CC --mode=link gcc -g _m2_pgeif.lo $(MY_DEPS) \
    pgeif_wrap.lo \
   -L$(GM2LIBDIR)/lib64 \
   -rpath `pwd` -liso -lgcc -lstdc++ -lpth -lc -lm -o libpgeif.la
cp .libs/libpgeif.so ../_pgeif.so
cp pgeif.py ../pgeif.py
```

More complex example

- passing data from Python into C, C++, Modula-2 shared library
  - can pass `int`, `float`, `double` and enums easily enough
- strings are also reasonably well supported
- how do we pass aggregate data types between Python and C/C++?
  - how do we return aggregate from C/C++ into Python?

Aggregate data types

- an aggregate data type is a data type which contains different sub types
  - for example a struct containing an int and a char field

- ```
  typedef struct aggregate_t {
    int field1;
    char field2;
  } aggregate;
  ```
  ```

Passing aggregate data types from Python into C/C++

- fortunately binary strings of data can be passed between Python and C/C++ using swig

- we can build a sequence of bytes using the Python `struct` module
    - the `struct` module uses a printf formatting structure to pack and unpack binary data

Why do we need to pass aggregate data types from C/C++ to Python?

- consider, `pge`, the shared library module generate events which might be:
  - a draw frame event
  - a collision event
  - a timer event

- the draw frame event
  - contains a list of polygons and circles and their position and colour which need to be rendered to represent the world
  - this is a dynamic list of objects containing many different data types

Why do we need to pass aggregate data types from C/C++ to Python?

- a collision event
  - contains the time of collision, position of the collision
  - and the object `ids` in collision

- this will be a fixed aggregate structure of known length

- the timer event will have a time field (`double`) and the timer `id` (`integer`) as well as a few other fields
  - this is also fixed in length and represented in C as a `struct`

Passing aggregate data from C, C++, Modula-2 into Python

- we can use the string passing mechanism to pass bytes
  - the `.i` file needs extra information to say which functions return binary data and also that the shared library can set the length

- **pge/i/pgeif.i**

```
...
%include cstring.i
%cstring_output_allocate_size(char **s, int *slen, );

%{
extern "C" void get_cbuf (char **s, int *slen);
extern "C" void get_ebuf (char **s, int *slen);
extern "C" void get_fbuf (char **s, int *slen);
...
```

Passing aggregate data from C, C++, Modula-2 into Python

- notice that a Python string is created in the shared library and passed back to the Python caller
- also notice that `get_cbuf` is a function!
  - returning a string
- the `swig` information
- ```
  %include cstring.i
  %cstring_output_allocate_size(char **s, int *slen, );
  ```
- indicates these types and name match a return string allocated in the shared library
  ```

Passing aggregate data from C, C++, Modula-2 into Python

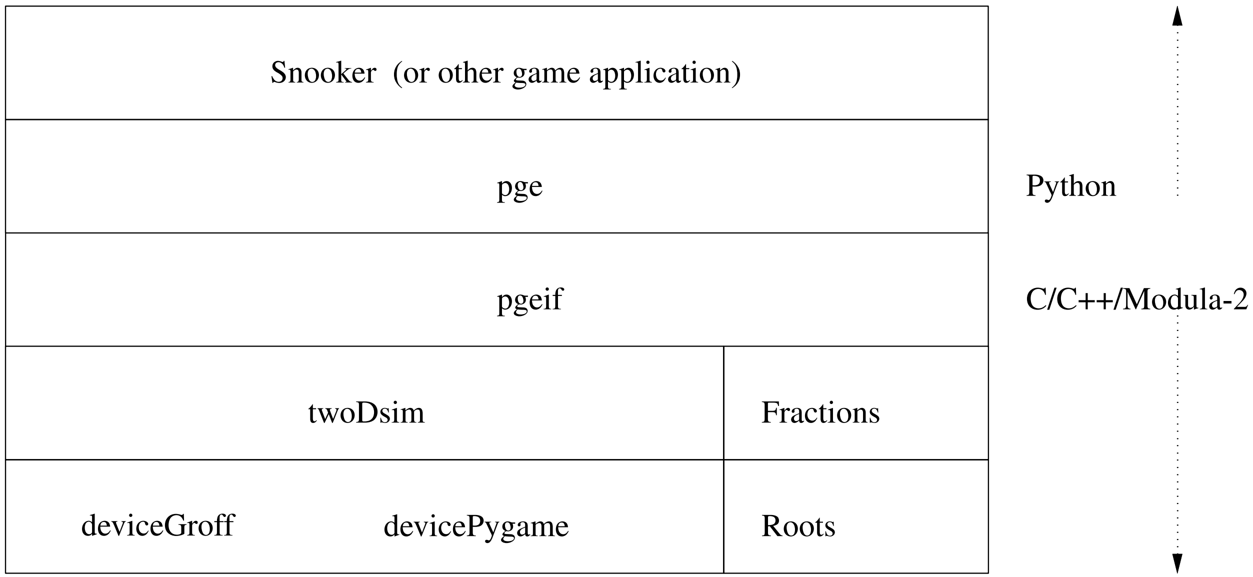
- **pge/python/pge.py**

```
def runbatch (t):
    if t < 0.0:
        t = 30.0
    _debugf ("runbatch (%f)\n", t)
    pgeif.check_objects ()
    cData = pgeif.get_cbuf ()
    fData = pgeif.get_fbuf ()
    _draw_frame (cData, len (cData), fData, len (fData))
    pgeif.empty_fbuffer ()
    pgeif.empty_cbuffer ()
```

Passing aggregate data from C, C++, Modula-2 into Python

- `swig` has many mechanisms to allow binary strings of data to be retrieved
  - above is the safest - as it contains the `length`

PGE structure



Passing aggregate data from C, C++, Modula-2 into Python

- examine the function `_draw_frame` which calls the function

- **pge/python/pge.py**

```
#
#   _pyg_draw_frame - draws a frame on the pygame display.
#

def _pyg_draw_frame (cdata, clength, fdata, flength):
    global nextFrame, call, _record
```

Passing aggregate data from C, C++, Modula-2 into Python

- **pge/python/pge.py**

```
    if _record:
        _begin_record_frame (cdata, clength, fdata, flength)
    elif flength > 0:
        _draw_background ()
    f = _myfile (cdata + fdata)
    while f.left () >= 3:
        header = struct.unpack ("3s", f.read (3))[0]
        header = header[:2]
        if call.has_key (header):
            f = call[header] (f)
        else:
            print "not understood header =", header
            sys.exit (1)
```

Passing aggregate data from C, C++, Modula-2 into Python

- **pge/python/pge.py**

```
    if flength > 0:
        _draw_foreground ()
    if _record:
        _end_record_frame ()
    if flength > 0:
        _doFlipBuffer ()  # flipping the buffer for an empty frame looks ugly
    nextFrame += 1
    _debugf ("moving onto frame %d\n", nextFrame)
```

Inside the shared library

- it creates the byte string containing aggregate data

Inside the shared library

pge/c/buffers.c

```
/*
 *  buffers - wrap the event buffer contents into a binary string.
 */

extern void deviceIf_getFrameBuffer (void **start,
                                     int *length, int *used);

void get_fbuf (void **start, unsigned int *used)
{
  int length;
#if !defined (DEBUGGING)
  printf ("calling deviceIf_getFrameBuffer\n");
#endif
  deviceIf_getFrameBuffer (start, &length, used);
}
```

Inside the shared library

- examine the file `pge/c/deviceIf.c`
  - follow the functions: `deviceIf_emptyFbuffer`, `deviceIf_useBuffer` and `deviceIf_finish`
- notice the use of the module `MemStream`
  - read the documentation of `MemStream` ⟨http://nongnu.org/gm2/gm2-libs-isomemstream.html⟩
- `MemStream` allows the caller to use file operations to maintain a byte string which is contiguous and held in memory

Conclusion and pgeif.i

- the full API describing the C interface is described in `pge/i/pgeif.i`
  - examine this file and see how a `circle`, `colour` and `box` are created

- now read the file `pge/python/pge.py` and see how a call to `box` and `colour` is mapped into the `pgeif.i` calls