

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, 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.o
$ libtool --tag=CC --mode=link gcc -g mymodule.o mymodule.o -L$HOME/opt/lib64 -rpath 'pwd' -lc -lm -o libmymodule.so
$ cp .libs/libmymodule.so _mymodule.so
$ 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 -o mymodule_wrap.lo -fPIC -DPIC
libtool: compile: gcc -g -I/usr/include/python2.7 -c mymodule_wrap.c -o mymodule_wrap.o -fPIC -DPIC -DPIC
libtool: compile: gcc -g -I/usr/include/python2.7 -c mymodule.c -o mymodule.o -fPIC -DPIC -DPIC
libtool: compile: gcc -g -I/usr/include/python2.7 -c mymodule.c -o mymodule.o -fPIC -DPIC -DPIC
libtool: link: rm -fr .libs/libmymodule.a .libs/libmymodule.la .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=libmymodule.so.0 -o .libs/libmymodule.so
libtool: link: (cd ".libs" && rm -f "libmymodule.so.0" && ln -s "libmymodule.so" "libmymodule.so.0")
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" )
```

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

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

- harder - but it can be done
 - the .i file needs extra information to say which functions give binary data and its length

```
%cstring_output_allocate_size(char **start, int *used);
%{
extern "C" void get_ebuf (char **start, int *used);
extern "C" void get_fbuf (char **start, int *used);
```

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

Case study: pge

```
#!/usr/bin/env python
import pge
print "starting exampleBoxes"
pge.batch ()
t = pge.rgb (1.0/2.0, 2.0/3.0, 3.0/4.0)
wood_light = pge.rgb (166.0/256.0, 124.0/256.0, 54.0/256.0)
wood_dark = pge.rgb (76.0/256.0, 47.0/256.0, 0.0)
metal = pge.rgb (0.5, 0.5, 0.5)
ball_size = 0.04
boarder = 0.01
```

Case study: pge

```
def placeBoarders (thickness, color):
    print "placeBoarders"
    e1 = pge.box (0.0, 0.0, 1.0, thickness, color).fix ()
    e2 = pge.box (0.0, 0.0, thickness, 1.0, color).fix ()
    e3 = pge.box (1.0-thickness, 0.0, thickness, 1.0, color).fix ()
    e4 = pge.box (0.0, 1.0-thickness, 1.0, thickness, color).fix ()
    for e in [e1, e2, e3, e4]:
        e.on_collision (play_wood)
    return e1, e2, e3, e4
```

Case study: pge

```
def crate (x, y, w):
    c = pge.box (x, y, w, w, wood_dark).on_collision (cra
```

Case study: pge

```
def crate_split (p):
    w = p.width () / 2
    wg = w - gap
    e = p.get_param ()
    if e != None:
        if e % 2 == 1:
            # subdivide into smaller crates, every odd bounce
            m = p[0].mass ()
            c = p[0].colour ()
            for v in [[0, 0], [0, w], [w, 0], [w, w]]:
                b = pge.box (v[0], v[1], wg, wg, c).mass (m).on_collision (crate)
                b.set_param (e-1)
            p.delete ()
        elif e == 0:
            # at the end of 6 collisions the crates disappear
            p.delete ()
        else:
            # allow collision (bounce) without splitting every even bounce
            p.set_param (e-1)
```

Overview of the Python interface to PGE

crate_example.py	application code
pge.py	Python API for PGE wraps low level object Python class (tagged)
pgeif.py	swig generated PGE interface - binary stream
pgeif.c	Modula-2 (or C, C++) code which translates PGE objects

Conclusion

- work in progress
- having an easy mechanism to write to a (char *) memory buffer is very useful (Modula-2 MemStream)
- struct module in Python is also very useful
- wrapping low level shared library objects (represented by ints) is also good
 - can test parameter types at run time in Python before they are translated into the pge int objects

Conclusion

```
swig -outdir . -o pgeif_wrap.cxx -c++ -python $(srcdir)/p
libtool --tag=CC --mode=compile g++ -g -c pgeif_wrap.cxx
-I/usr/include/python$(PYTHON_VERSION) -o pgeif_wrap.lo
gm2 -c -g -I$(SRC_PATH_PIM) -fmakelist $(srcdir)/pgeif.mo
gm2 -c -g -I$(SRC_PATH_PIM) -fmakeinit -fshared $(srcdir)
libtool --tag=CC --mode=compile g++ -g -c _m2_pgeif.cpp -
libtool --tag=CC --mode=link gcc -g _m2_pgeif.lo $(PGELIB
pgeif_wrap.lo buffers.lo \
-L$(prefix)/lib64 \
-rpath `pwd` -lgm2 -liso -lgcc -lstdc++ -lpth -lc -lm -
cp .libs/libpgeif.so _pgeif.so
```