Compiling, linking, and language-mixing

Andy Buckley
University of Glasgow

MCnet Computing School, Mariaspring, 19 May 2016

Preprocessing, compilation, and linking

Building C, C++, Fortran, etc. is split into several stages:

- **Preprocessing:** dumb text concatenation and replacement. Optional extension for Fortran. Pass `-E` to gcc to stop build at this point. `cpp` command, `CPPFLAGS` env vars
- **Compiling:** convert high-level language into CPU-specific instruction set (assembly language). Pass `-S` to stop. `CFLAGS`, `FFLAGS`, `CXXFLAGS` env vars
- **Assembly:** convert assembly code to binary. Pass `-c`. Access *object file* content with `nm`
- **Linking:** put together binary object files with startup code and *libraries*. Knows nothing about code, argument lists, etc.
- For small programs usually run all steps from one `gcc/g++/gfortran` command

Object files and libraries

- EXERCISE: build **`cdemo.c`**, **`ccdemo.cc`**, **`fdemo.f90`** →**`*demo.o`** and view contents with **`nm`**
- The symbols' binary contents are independent of language, but their *names* are not
  - "Name mangling": C = no-mangling, Fortran unstandardised without effort, C++ unstandardised
  - But if we are careful, we can make languages talk to each other...
- To build a *static* library of combined object files, use **`ar`**:

  `ar -rcs libmy.a *demo.o`
- To build a *shared/dynamic* library of combined object files, use **`gcc`**:

  `gcc -shared -o libmy.so *demo.o`
- Try **`objdump -p`**, **`readelf -a`** to inspect the resulting object/lib files. Note **`-C`** option or explicit **`c++filt`** for decoding of C++ name mangling.

Static and dynamic linking

- Linking: `g++ foo.cc -o foo /prefix/lib/libmy.so` or `g++ foo.cc -o foo -L/prefix/lib -lmy`
- Static libs are just big object files with a global symbol table
- Linking an executable against a static lib pulls all symbols into the executable
  - Can be convenient, fast, portable, no path issues – but very large executables and dependency/relink nightmares
  - Lib/object file order on linker command line matters (argh)

Static and dynamic linking

- Linking: `g++ foo.cc -o foo /prefix/lib/libmy.so` or `g++ foo.cc -o foo -L/prefix/lib -lmy`
- Shared libs are cleverer, do dynamic lookup: more metadata, use `ld` at runtime $\Rightarrow$ `(DY)LD_LIBRARY_PATH`
  - Keeps executables small, automatic dependency resolution as long as no API/ABI change
  - Standard for system applications, and more modern physics code
  - Runtime sensitivity: `ldd` can be used to work out current environment's path resolution for an executable or shared lib

Calling Fortran from C(++)

- Use `extern "C"` blocks to get C non-mangling
- Forward-declare Fortran functions as symbols with case and trailing underscore(s) mangling
- Use! But several mapping details to be careful about:
  - Args and returns: passed as pointers, cannot be literals; subroutines return into args
  - Type matching and symbol manglings are not standard, may be compiler dependent; F2003 adds better compatibility mechanisms
  - Some features/types may not map well: use *shim functions* in either lang to provide a clean interface
  - String args are fiddly: try to avoid! (in Fortran: no null-termination, secret non-std extra arg for lengths)
  - Note that array indexing is inverted, so Fortran `INTEGER(1,2,3)` $\rightarrow$ C(++) `int[3][2][1]`
  - Remember to explicitly link in the *other* std library: `-lstdc++` or `-lgfortran`

Calling C(++) from Fortran

- You can't call all C++ features from Fortran – classes have no direct equivalent, for example
- But you can call C-linkage C++ functions which internally use objects
- Shim functions can be written in either C++ or Fortran
- Much easier to map C++ functions into Fortran subroutines than into Fortran functions
- Otherwise it's just the same as Fortran from C(++)

- **In both directions, keep it as simple as possible: this stuff needs to be functional, not beautiful**

Calling C from Python: `ctypes`

- Easiest, but not "native": `ctypes` module
- EXAMPLE:

```
import ctypes
demo = ctypes.CDLL("libdemo.so")
demo.fib(5)
```

- For floating point types, need to specify argument and return types by hand:

```
demo.dbldbl(4.0) # error!!
demo.dbldbl.argtypes = [ctypes.c_double]
demo.dbldbl.restype = ctypes.c_double
demo.dbldbl.argtypes(4.0) # 8.0
```

- Calling C++ or Fortran requires knowing the symbol mangling

Calling C(++) from Python: `Cython`

- Cython can also be used to access C(++) code – classes and functions – from Python
  - Originally developed (as Pyrex) as a typed Python-like language: also useful for optimisation, after comprehensions and numpy
  - Can be used to make "native-looking" Python C extensions: docstrings, auto-completion, customised to be Pythonic if wanted
- EXAMPLE: see `cydemo.pyx` and `cydemo.pxd`

```
cython --cplus cydemo.pyx
g++ -c -fPIC cydemo.cpp
gcc -shared -o cydemo.so cydemo.o libdemo.so
```

- Nicely integrated (less manual) builds via Python distutils – see the excellent docs
- See also SWIG for simpler, but more automated mapping (and other scripting languages)