The Abdus Salam International Centre for Theoretical Physics

Compiling, Linking & Mix Languages

Ivan Girotto – igirotto@ictp.it

International Centre for Theoretical Physics (ICTP)

Script Language Benefits

- Portability
  - Script code does not need to be recompiled
  - Platform abstraction is part of script library
- Flexibility
  - Script code can be adapted much easier
  - Data model makes combining multiple extensions easy
- Convenience
  - Script languages have powerful and convenient facilities for pre- and post-processing of data
  - Only time critical parts in compiled language

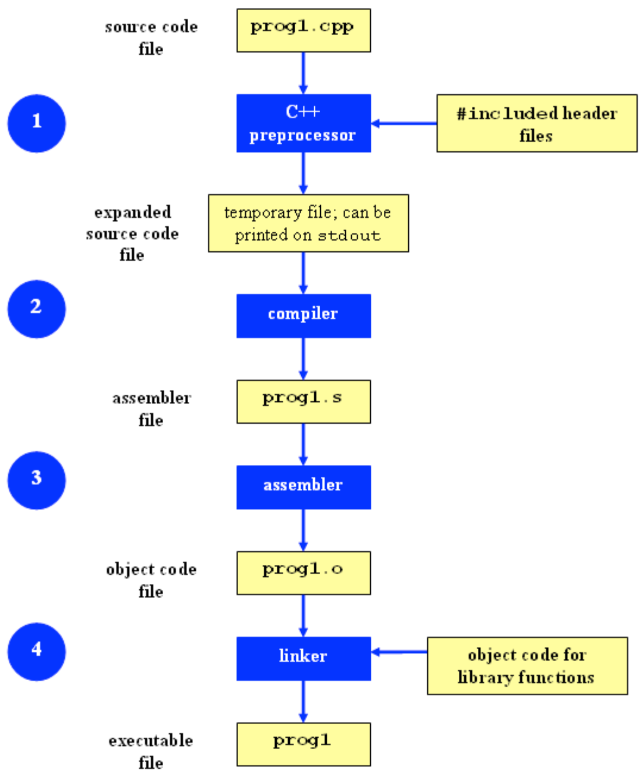
From Scripting to Compiled Codes

- maximum control of the low-level implementation
- high-performance
  - compiler are written to deliver best optimization by having full/relevant knowledge of the back-end architecture
- the O.S. loads the binary into memory and starts the execution (no other support would be required)
- direct interface to most of scientific code available

The Compiler

- Creating an executable includes multiple steps
- The “compiler” (gcc) is a wrapper for several commands that are executed in succession
- The “compiler flags” similarly fall into categories and are handed down to the respective tools
- The “wrapper” selects the compiler language from source file name, but links “its” runtime
- We will look into a C example first, since this is the language the OS is (mostly) written in

The Compiling Phases




```
#include <stdio.h>
int main(int argc, char **argv)
{
    printf("hello world\n");
    return 0;
}
```

Compilation Command examples

Pre-Processing

- Pre-processing is mandatory in C (and C++)
- Pre-processing will handle '#' directives
  - File inclusion with support for nested inclusion
  - Conditional compilation and Macro expansion
- In this case: **/usr/include/stdio.h**
  - and all files are included by it - are inserted and the contained macros expanded
- Use -E flag to stop after pre-processing:
  - **gcc -E -o hello.pp.c hello.c**
  - **cpp main.c main.i (same)**

Compiling

- Compiler converts a high-level language into the specific instruction set of the target CPU
- Individual steps:
  - Parse text (lexical + syntactical analysis)
  - Do language specific transformations
  - Translate to internal representation units (IRs)
  - Optimization (reorder, merge, eliminate)
  - Replace IRs with pieces of assembler language
- Using -S the compilation stops after the stage of compilation (does not assemble). The output is in the form of an assembler code file for each non-assembler input file specified.
  - **gcc -S hello.c (produces hello.s)**

Assembling

- Assembler (as) translates assembly to binary
  - from there, Linux tools are needed for accessing the content
- Creates so-called object files (in ELF format)
  - gcc -c hello.c
  - nm hello.o
- Be careful at *built-in* functions
  - -fno-builtin can be used to work-around the problem

Linking

- Linker (ld) puts binary together with startup code and required libraries
- Final step, result is executable
  - gcc -o hello hello.o
- The linker then “builds” the executable by matching undefined references with available entries in the symbol tables of the objects/libraries

Why is a linker interesting to us?!

- Understanding linkers will help you to build large programs
- Understanding linkers will help you to avoid dangerous programming errors
- Understanding linkers will help you how language scoping rules are implemented
- Understanding linkers will help you understand how things works
- Understanding linkers will enable you to exploit shared libraries

Object Files

- Object Files are divided in three categories:
  - Rolocatable Object Files (*.o)
  - Executable Object File
  - Shared Object Files
- Compiled object files have multiple sections and a symbol table describing their entries:
  - “Text”: this is executable code
  - “Data”: pre-allocated variables storage
  - “Constants”: read-only data
  - “Undefined”: symbols that are used but not defined
  - “Debug”: debugger information (e.g. line numbers)
- Sections can be inspected with the “readelf” command

Symbols in Object Files

```
ig@hp83-inf-21> nm visibility.o
0000000000000000 t add_abs
000000000000002a T main
                 U printf
0000000000000000 r val1
0000000000000004 R val2
0000000000000000 d val3
0000000000000004 D val4
```

```c
#include <stdio.h>

static const int val1 = -5;
const int val2 = 10;
static int val3 = -20;
int val4 = -15;
extern int errno;

static int add_abs(const int v1, const int v2) {
  return abs(v1)+abs(v2);
}

int main(int argc, char **argv) {

  int val5 = 20;

  printf("%d / %d / %d\n",
         add_abs(val1,val2),
         add_abs(val3,val4),
         add_abs(val1,val5));

  return 0;
}
```

Static Libraries

- Static libraries built with the “ar” command are collections of objects with a global symbol table
- When linking to a static library, object code is copied into the resulting executable and all direct addresses recomputed (e.g. for “jumps”)
- Symbols are resolved “from left to right”, so circular dependencies require to list libraries multiple times or use a special linker flag
- When linking only the name of the symbol is checked, not whether its argument list matches

```
#building static the library
ig@hp83-inf-21 > ar -rcs libmy.a myfile*.o


#brute force linking
ig@hp83-inf-21 > gcc main.c ./libmy.a


#Using -L (tells the compiler where look for libraries)
ig@hp83-inf-21 > gcc main.c -L./ -lmy


#Same above using gcc notation
igi@hp83-inf-21 > gcc main.c \
> -Wl,--library-path=/scratch/igirotto/linking -Wl,-lmy
```

Shared Libraries

- Shared libraries are more like executables that are missing the main() function
- When linking to a shared library, a marker is added to load the library by its "generic" name (soname) and the list of undefined symbols
- When resolving a symbol (function) from shared library all addresses have to be recomputed (relocated) on the fly.
- The shared linker program is executed first and then loads the executable and its dependencies

```
#building shared library
ig@hp83-inf-21 > gcc -shared -o mylib.so swap.o

#brute force linking
ig@hp83-inf-21 > gcc main.c ./libmy.so

#Using -L (tells the compiler where look for libraries)
ig@hp83-inf-21 > gcc main.c -L./ -lmy
ig@hp83-inf-21 > ldd a.out
   linux-vdso.so.1 =>  (0x00007fffdbb6b000)
   libmy.so => not found
   /lib64/ld-linux-x86-64.so.2 (0x00007fa003cd1000)

#Add a directory to the runtime library search
pathigi@hp83-inf-21 > gcc main.c \
> -Wl,--rpath=/scratch/igirotto/linking -Wl,-lmy
```

Using LD_PRELOAD

- Using the LD_PRELOAD environment variable, symbols from a shared object can be preloaded into the global object table and will override those in later resolved shared libraries
  - replace specific functions in a shared library
- Example: override log() with a faster version:

```
double log(double x) {
   return my_log(x);
}
```

```
$gcc -shared -o fasterlog.so faster.c -lmy_log
$LD_PRELOAD=./fasterlog.so ./myprog-with
```

Mixed Linking

- Fully static linking is a bad idea with GNU libc; it requires matching shared objects for NSS
- Dynamic linkage of add-on libraries requires a compatible version to be installed (e.g. MKL)
- Static linkage of individual libs via linker flags -Wl,-Bstatic,-lfftw3,-Bdynamic
- can be combined with grouping, example:
  - gcc [...] -Wl,--start-group,-Bstatic -lmkl_gf_lp64 \
    -lmkl_sequential -lmkl_core -Wl,--end-group,-Bdynamic

From C to FORTRAN

- Basic compilation principles are the same
  - preprocess, compile, assemble, link
- In Fortran, symbols are case insensitive
  - most compilers translate them to lower case
- In Fortran symbol names may be modified to make them different from C symbols (e.g. append one or more underscores)
- Fortran entry point is not “main” (no arguments) PROGRAM => MAIN__ (in gfortran)
- C-like main() provided as startup (to store args)

Symbols in Object Files (FORTRAN COMPILED)

```
ig@hp83-inf-21> nm test.o
000000000000006d t MAIN__
                 U
_gfortran_set_args
                 U
_gfortran_set_options
                 U
_gfortran_st_write
                 U
_gfortran_st_write_done
                 U
_gfortran_transfer_character_write
0000000000000000 T greet_
0000000000000078 T main
0000000000000020 r options.1.1883
```

```
SUBROUTINE GREET
  PRINT*, 'HELLO, WORLD!'
END SUBROUTINE GREET
program hello
  call greet
end program
```

Fortran 90+ Modules

```
module func
      integer :: val5, val6
   contains
      integer function add_abs(v1,v2)
            integer, intent(in) :: v1, v2
            add_abs = iabs(v1)+iabs(v2)
      end function add_abs
end module func
```

gfortran creates the following symbols:

```
00000000 T __func_MOD_add_abs
00000000 B __func_MOD_val5
00000004 B __func_MOD_val6
```

Dynamic Linking via dlopen()

- POSIX compliant C libraries allow loading of shared objects are runtime via dlopen()/dlsym()
- Calls to dlopen() open a handle to shared object; lookup of this file is subject to same rules as dynamic library searches
- Calls to dlsym() look up symbol by its name in shared object pointed to by handle; returns pointer; for functions need to cast/assign to function pointer
- Calls to dlclose() unload shared object (if last user) and revoke assignments to code made by dlsym()

Example: static program test-0.c

```
#include <stdio.h>
void hello()
    puts("Hello, World");
}
int main(int argc, char **argv)
{
    void (*hi)(); /* function pointer variable */
    hi = &hello; /* initialize function pointer */
    (*hi)(); /* this is the same as: hello(); */
    return 0;
}
/* compile with: gcc -o test-0 -Wall -O test-0.c */
```

Example: main program test-1.c

```
#include <dlfcn.h>
int main(int argc, char **argv)
{
    void *handle; /* handle for dynamic object */
    void (*hi)(); /* function pointer for symbol */
    handle = dlopen("./hello.so", RTLD_LAZY);
    if (handle) {
        hi = (void (*)()) dlsym(handle,"hello");
        (*hi)();
        dlclose(handle);
    }
    return 0;
}/* compile with: gcc -o test-1 -Wall -O test-1.c -ldl
    add -rdynamic if shared object needs symbols in main */
```

Example: shared object hello.c

```
#include <stdio.h>
void hello(void)
{
   puts("Hello, World!");
}
/* compile: gcc -shared -o hello.so -fPIC -Wall -O hello.c */
```

- With this setup, hello.c can be changed and hello.so recompiled without having to recompile and re-link test-1
- Thus access to test-1.c is not needed

Extending Python with ctypes

- The ctypes module in python provides an interface to dlopen()/dlsym() and thus allows to call compiled C code from python.
- Support for dll files on Windows is also included
- Since symbols in compiled objects have no information about calling sequence and return values, this has to be set on the python side
- Incorrect use can lead to segmentation faults or corrupted data; often prototypes are needed

Example: calling hello.c from python

```
#!/usr/bin/env python
from ctypes import *
# import shared object on POSIX compatible OS
dso = CDLL(".̀/hello.so")
# call symbol in shared object as function w/o args
dso.hello()
```

- This python script does pretty much the same thing as the test-1 compiled program
- Since there are no arguments and no return values, no code needs to know about the other

Arguments & Return Value

- By default ctypes will assume arguments and return values are standard size integer

```
#include<stdio.h>
int sum_of_int(int a, int b) {
    int c = a + b;
    printf("sum of %d and %d is %d\n",a,b,c);
    return c;
}
```

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./sum.so”)
isum = dso.sum_of_int(1,2)
print "Integer sum is: ", isum
```

Prototypes with ctypes

- If argument and/or return value are of different type, ctypes needs to be informed about it; works similar to prototypes in C

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./sum.so”)
dso.sum_of_int.argtypes = [ c_int, c_int ]
dso.sum_of_int.restype = c_int
isum = dso.sum_of_int(1,2)
print ("Integer sum w/ prototypes is: ", isum)
dso.sum_of_double.argtypes = [ c_double, c_double ]
dso.sum_of_double.restype = c_double
dsum = dso.sum_of_double(0.5,2.5)
print ("Double sum w/ prototypes is: ", dsum)
```

Passing Strings

- Strings in python are read-only, thus when a C-function will modify a string we have to use create_string_buffer()

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL("./hello.so")
# hello() in hello.so takes a "char *" argument
dso.hello.argtypes = [ c_char_p ]
dso.hello(b"World")
# create buffer for mutable string data
buf = create_string_buffer(b"World")
dso.hello(buf)
```

Passing Arrays

- When passing allocatable objects like arrays, it is usually best to do the allocating in python. ctypes offers constructors for all basic types

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./sum.so”)
num = 10
dlist = (c_double * num)() # (primitive * length)()
for i in range(num):
dlist[i] = 0.333*(i*0.5)
# note the use of POINTER()
dso.sum_of_doubles.argtypes=[POINTER(c_double),c_int]
dso.sum_of_doubles.restype = c_double
dsum = dso.sum_of_doubles(dlist,num)
print ("Double sum is: ", dsum)
```

Passing Structs /1

- Even complex storage elements like struct can be managed by ctypes. Derive a class from Structure that mimics the corresponding C-type

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./data.so”)
class parm(Structure):
_fields_ = [ ("type", c_int), ("label", c_char_p),
("epsilon",c_double),("sigma",c_double) ]
# use constructor to initialize struct
p = parm(type=1,label=b"LJ-12-6",epsilon=0.1,sigma=3.4)
# p is passed by value, to pass by reference use byref(p)
dso.pass_by_value(p)
dso.pass_by_reference(byref(p))
```

Passing Structs /2

- Below is the corresponding C code:

```
#include<stdio.h>
struct parm { int type; char *label;
    double epsilon, sigma;
};
void pass_by_value(struct parm p) {
    printf("type=%d label=%s epsilon=%g sigma=%g\n",
           p.type, p.label, p.epsilon, p.sigma);
}
void pass_by_reference(struct parm *p) {
    printf("type=%d label=%s epsilon=%g sigma=%g\n",
           p->type, p->label, p->epsilon, p->sigma);
}
```

Interfacing Fortran with f2py /1

- Interfacing Fortran with python is both easier and more complicated than interfacing C
  - The Fortran ABI can be much more complex and is more compiler specific than the C ABI
  - The numpy project has a tool “f2py” that automates the process and hides the complications
- If you have a Fortran file with some functions or subroutine do: f2py -c code.f90 -m module
  - Creates python loadable module “module”
  - Flag '-c' calls compiler; flag '-m' sets module name

Interfacing Fortran with f2py /2

- Then in python do: from module import * and call the Fortran functions in python
- The f2py tool will parse the Fortran code and generate the necessary C-code for a module
- The f2py generated code will automatically insert code to convert data as needed; e.g. lists are converted to arrays
- The f2py tool works best with well formed Fortran code; otherwise data maps can help

- Example of Fortran code that converts cleanly with f2py:

```
subroutine hello
    print*, "Hello, World!"
end subroutine hello
function sum_of_int(a,b) result(c)
    integer, intent(in) :: a, b
    integer :: c
    c = a + b
    print*,"sum of ", a, " and ", b, " is ", c
end function sum_of_int
function sum_of_double(a,b) result(c)
    double precision, intent(in) :: a, b
    double precision :: c
    c = a + b
    print*,"sum of ", a, " and ", b, " is ", c
end function sum_of_double
```

Passing arrays with f2py

- Arrays are traditional style arrays with f2py:

```
function sum_of_doubles(a,n) result(s)
    double precision, intent(in) :: a(*)
    integer, intent(in) :: n
    double precision :: s
    integer :: i
    s = 0
    do i=1,n
        s = s + a(i)
    end do
end function sum_of_doubles
```

```
num = 10
dlist = [sqrt(float(i)) for i in range(1,num)]
dsum = sum_of_doubles(dlist,num)
```

Passing strings with f2py

- Strings are handled in a very similar fashion to traditional style arrays with f2py:

```
subroutine hello(name)
    character(len=*), intent(in) :: name
    print* , "Hello, ", name, "!"
end subroutine hello
```

```
#!/usr/bin/env python
from hello import *
print ("Calling DSO")
hello('World')
print ("Done")
```