Compiling, Linking & Mixed Languages

Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)

International Centre for Theoretical Physics (ICTP)

Embedded Scripting Language

- Not a new idea, but many scientific tools with scripting have their own “language”
  - script capability added on top of the tool
- Better to add domain specific extensions to an existing, generic scripting language:
  - use a language designed for scripting
  - can import other extensions, if needed
  - better documentation for script language
  - users may already know the syntax
- we have been using Python in this workshop

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

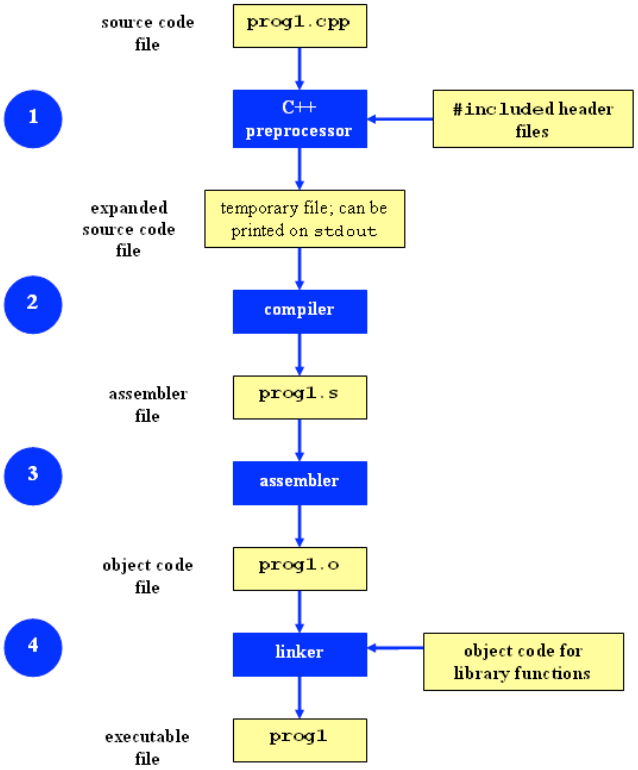
From Scripting to Compiled Codes

- 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

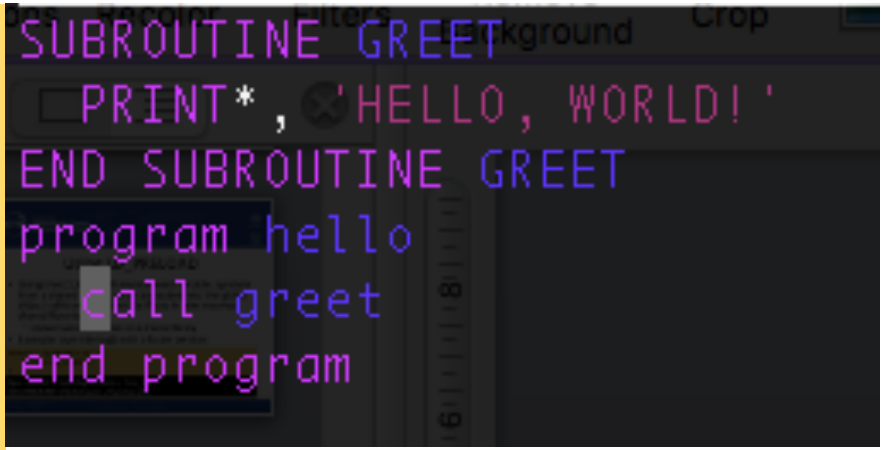
- Basic compilation principles are the same
  - preprocess, compile, assemble, link
- In Fortran, symbols are case insensitive
  - most compilers translate them to lower case
- In Fortran symbol names may be modified to make them different from C symbols (e.g. append one or more underscores)
- Fortran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

Pre-Processing in FORTAN

- Pre-processing is mandatory in C/C++
- Pre-processing is optional in Fortran
- Fortran pre-processing enabled implicitly via file name: name.F, name.F90, name.FOR
- Legacy Fortran packages often use /lib/cpp:
  - /lib/cpp -C -P -traditional -o name.f name.F
    - -C : keep comments (may be legal Fortran code)
    - -P : no '#line' markers (not legal Fortran syntax)
    - -traditional : don't collapse whitespace (incompatible with fixed format sources)

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
```



Symbols in C++

- In C++ functions with different number or type of arguments can be defined (overloading)
  - encode prototype into symbol name:
- Example : symbol for int add_abs(int,int) becomes: _ZL7add_absii
- Note: the return type is not encoded
- C++ symbols are no longer compatible with C
  - add 'extern “C”' qualifier for C style symbols
- C++ symbol encoding is compiler specific

Modern Fortran vs C Interoperability

- Fortran 2003 introduces a standardized way to tell Fortran how C functions look like and how to make Fortran functions have a C-style ABI
- Module “iso_c_binding” provides kind definition: e.g. C_INT, C_FLOAT, C_SIGNED_CHAR
- Subroutines can be declared with “BIND(C)”
- Arguments can be given the property “VALUE” to indicate C-style call-by-value conventions
- String passing tricky, needs explicit 0-terminus

Calling Fortran 03 From C Example

```
int sum_abs(int *in, int *num) {
        int i,sum;
        for (i=0,sum=0;i<*num;++i) {sum += abs(in[i]);}
    return sum;
}

/* fortran code:
        use iso_c_binding, only: c_int
        interface
                integer(c_int) function sum_abs(in, num) bind(C)
                        use iso_c_binding, only: c_int
                        integer(c_int), intent(in) :: in(*)
                        integer(c_int), value :: num
                end function sum_abs
        end interface
        integer(c_int), parameter :: n=200
        integer(c_int) :: data(n)
        print*, SUM_ABS(data,n) */
```

Calling Fortran 03 From C Example

```
subroutine sum_abs(in, num, out) bind(c)
     use iso_c_binding, only : c_int
     integer(c_int), intent(in) :: num,in(num)
     integer(c_int), intent(out) :: out
     integer(c_int), :: i, sum
     sum = 0
     do i=1,num
          sum = sum + ABS(in(i))
     end do
     out = sum
end subroutine sum_abs

!! c code:
! const int n=200;
! int data[n], s;
! sum_abs(data, &n, &s);
! printf("%d\n", s);
```

Linking Multi-Language Binaries

- Inter-language calls via mutual C interface only due to name “mangling” of C++ / Fortran 90+
  - extern “C”, ISO_C_BINDING, C wrappers
- Fortran “main” requires Fortran compiler for link
- Global static C++ objects require C++ for link
  - avoid static objects (good idea in general)
- Either language requires its runtime for link
  - GNU: -lstdc++ and -lgfortran
  - Intel: “its complicated” (use -# to find out) more may be needed (-lgomp, -lpthread, -lm)

Thanks for your attention!!