Linking with libraries using multiple programming languages

Dr. Axel Kohlmeyer

Senior Scientific Computing Expert

Information and Telecommunication Section
The Abdus Salam International Centre
for Theoretical Physics

http://sites.google.com/site/akohlmey/

akohlmey@ictp.it

The Abdus Salam International Centre for Theoretical Physics

Workshop on Computer Programming and Advanced Tools for Scientific Research Work

Symbols in Object Files & Visibility

- 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)
- Entries in the object files can be inspected with either the “nm” tool or the “readelf” command

Example File: visbility.c

```
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;
}
```

```
nm visibility.o:
00000000 t add_abs
         U errno
00000024 T main
         U printf
00000000 r val1
00000004 R val2
00000000 d val3
00000004 D val4
```

The Abdus Salam International Centre for Theoretical Physics

What Happens During Linking?

- Historically, the linker combines a “startup object” (crt1.o) with all compiled or listed object files, the C library (libc) and a “finish object” (crtn.o) into an executable (a.out)
- Nowadays it is more complicated
- The linker then “builds” the executable by matching undefined references with available entries in the symbol tables of the objects
- crt1.o has an undefined reference to “main” thus C programs start at the main() function

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

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

Differences When Linking

- Static libraries are fully resolved “left to right”; circular dependencies are only resolved between explicit objects or inside a library -> need to specify librariess multiple times or use: **-Wl,--start-group (...) -Wl,--end-group**
- Shared libraries symbols are **not** fully resolved at link time, only checked for symbols required by the object files. Full check only at runtime.
- Shared libraries may depend on other shared libraries whose symbols will be globally visible

Semi-static Linking

- Fully static linkage is a bad idea with glibc; <u>requires</u> 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:

```
-Wl,--start-group,-Bstatic                  \
      -lmkl_gf_lp64 -lmkl_sequential        \
      -lmkl_core -Wl,--end-group,-Bdynamic
```

The Abdus Salam International Centre for Theoretical Physics

Dynamic Linker Properties

- Linux defaults to dynamic libraries:
  `> ldd hello`
  `linux-gate.so.1 =>  (0x0049d000)`
  `libc.so.6 => /lib/libc.so.6 (0x005a0000)`
  `/lib/ld-linux.so.2 (0x0057b000)`
- **`/etc/ld.so.conf, LD_LIBRARY_PATH`** define where to search for shared libraries
- **`gcc -Wl,-rpath,/some/dir`** will encode **`/some/dir`** into the binary for searching

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() and exp() in libm:

```
#include "fastermath.h"
double log(double x) { return fm_log(x); }
double exp(double x) { return fm_exp(x); }
```

- gcc -shared -o faster.so faster.c -lfastermath
- LD_PRELOAD=./faster.so ./myprog-with

The Abdus Salam International Centre for Theoretical Physics

Difference Between C and Fortran

- Basic compilation principles are the same
  => preprocess, compile, assemble, link
- In Fortran, symbols are <u>case insensitive</u>
  => most compilers <u>translate</u> 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)

The Abdus Salam
International Centre
for Theoretical Physics

Fortran Example

```
SUBROUTINE GREET
  PRINT*, 'HELLO, WORLD!'
END SUBROUTINE GREET

program hello
  call greet
end program
```

```
0000006d t MAIN__
         U _gfortran_set_args
         U _gfortran_set_options
         U _gfortran_st_write
         U _gfortran_st_write_done
         U _gfortran_transfer_character
00000000 T greet_
0000007a T main
```

- "program" becomes symbol "MAIN__" (compiler dependent)
- "subroutine" name becomes lower case with '_' appended
- several "undefineds" with '_gfortran' prefix
  => calls into the Fortran runtime library, libgfortran
- cannot link object with "gcc" alone, need to add -lgfortran
  => cannot mix and match Fortran objects from different compilers

Fortran 90+ Modules

- When subroutines or variables are defined inside a module, they have to be hidden

```
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
```

The Abdus Salam International Centre for Theoretical Physics

The Next Level: 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 <u>not</u> encoded
- C++ symbols are no longer compatible with C => add 'extern “C”' qualifier for C style symbols
- C++ symbol encoding is <u>compiler specific</u>

The Abdus Salam International Centre for Theoretical Physics

C++ Namespaces and Classes vs. Fortran 90 Modules

- Fortran 90 modules share functionality with classes and namespaces in C++
- C++ namespaces are encoded in symbols
  Example: `int func::add_abs(int,int)`
  becomes: `_ZN4funcL7add_absEii`
- C++ classes are encoded the same way
- Figuring out which symbol to encode into the object as undefined is the job of the compiler
- When using the gdb debugger use '::' syntax

Why We Need Header or Module Files

- The linker is "blind" for any language specific properties of a symbol => checking of the validity of the interface of a function is only possible during compilation
- A header or module file contains the prototype of the function (not the implementation) and the compiler can compare it to its use
- Important: header/module has to match library => Problem with FFTW-2.x: cannot tell if library was compiled for single or double precision

Calling C from Fortran

- Need to make C function look like Fortran 77 => provide a wrapper function visible in Fortran
  - Append underscore
  - Use call by reference conventions
  - Best only used for “subroutine”

    ```
    void add_abs_(int *v1,int *v2,int *res){
    *res = abs(*v1)+abs(*v2);}
    ```
- Arrays are always passed as flat arrays
- String passing is tricky (no zero-termination) (length typically appended to list of arguments)

Calling C from Fortran Example

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

/* fortran code:
   integer, parameter :: n=200
   integer :: s, data(n)

   call SUM_ABS(data, n, s)
   print*, s
*/
```

The Abdus Salam International Centre for Theoretical Physics

Calling Fortran from C

- Inverse from above, i.e. need to add underscore and use lower case
- Difficult for anything but Fortran 77 style calls since Fortran 90+ features need extra info
  - Shaped arrays, optional parameters, modules
- Arrays need to be “flat”,
  C-style multi-dimensional arrays are lists of pointers to individual pieces of storage, which may not be consecutive
  => use 1d and compute position

Calling Fortran From C Example

```
subroutine sum_abs(in, num, out)
   integer, intent(in)   :: num, in(num)
   integer, intent(out)  :: out
   Integer               :: 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);
```

The Abdus Salam International Centre for Theoretical Physics