Environment Canada Environnement Canada

Canada

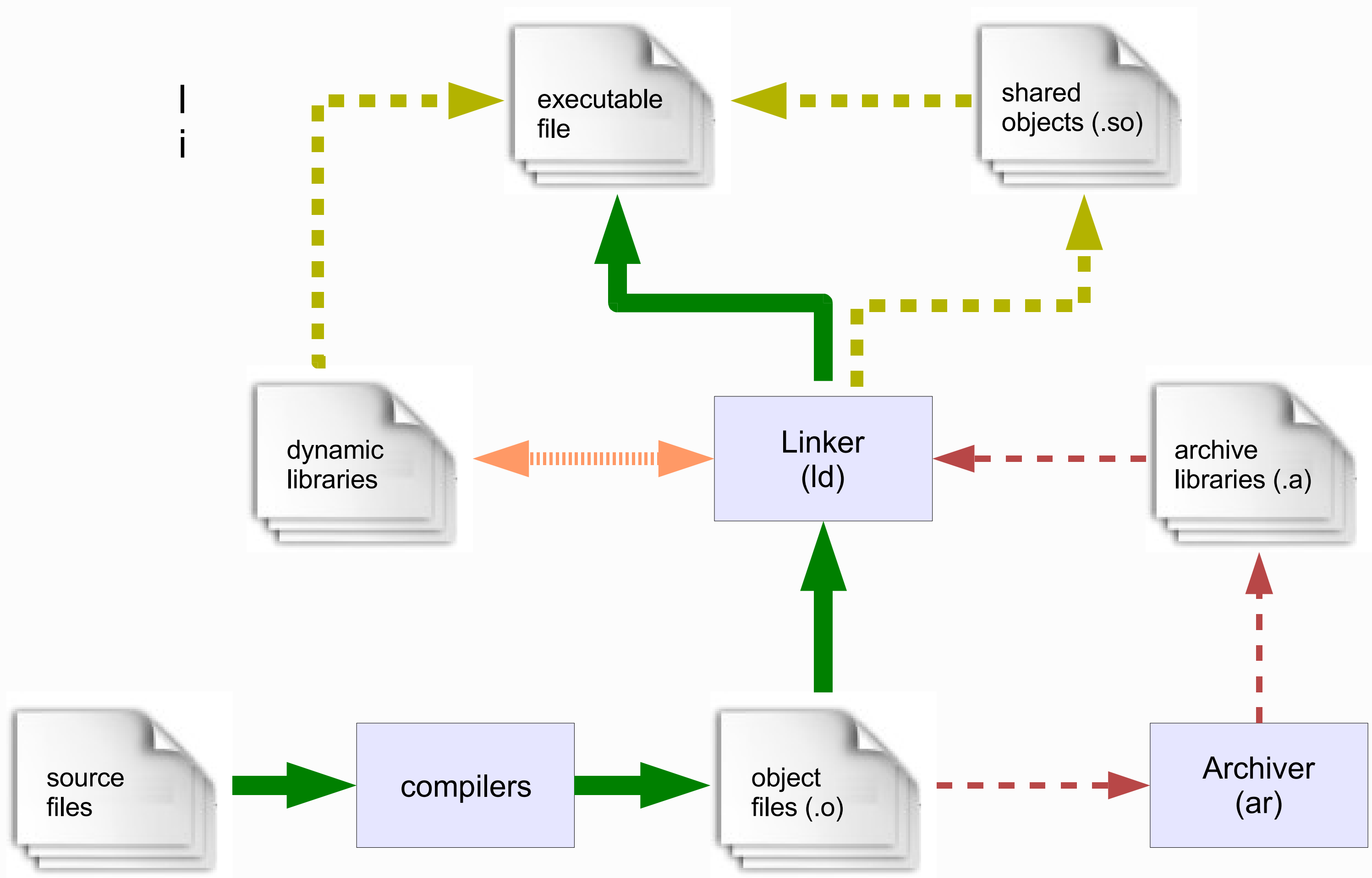
Compilers and executables
(the dark side of the force)

Michel Valin
CIOB / HPCS

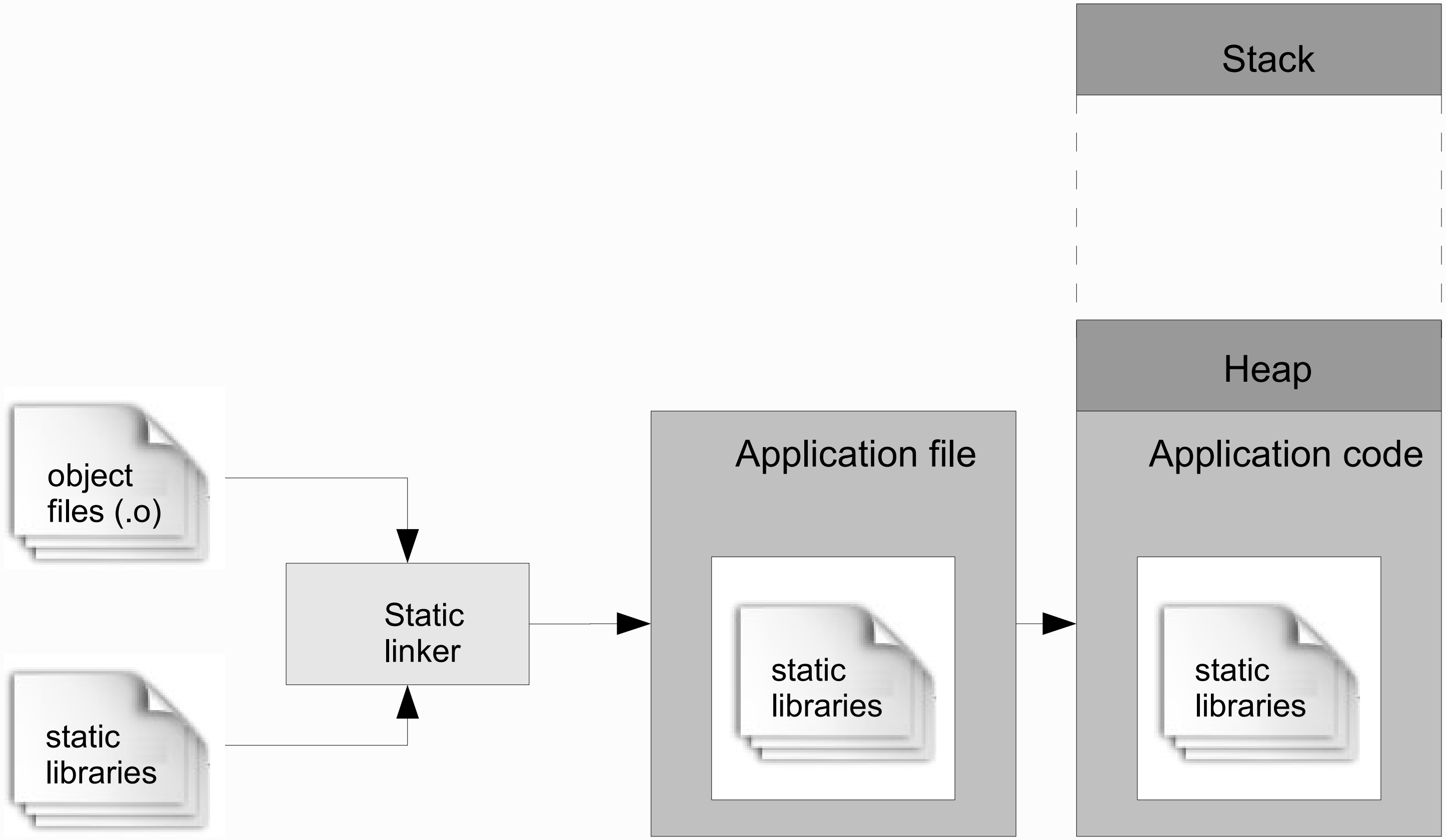
Tools

- The basic tools
  - Compilers (cc, CC, f77, f90, …)
    - Transforms source code into machine code
  - Linker (ld)
    - Builds an executable file
  - Library manager (ar)
    - Allows to maintain ordered sets of machine code objects
  - Symbol lister (nm)
    - Lists symbols defined and used in object code
  - Library dependency lister (ldd)
    - Finds which libraries will be used at run time
  - Symbol remover (strip)
    - Make an object smaller by removing debug and info symbols
  - Find strings in object (strings)
    - Extract strings (C style, null terminated) from binary file

The code labyrinth



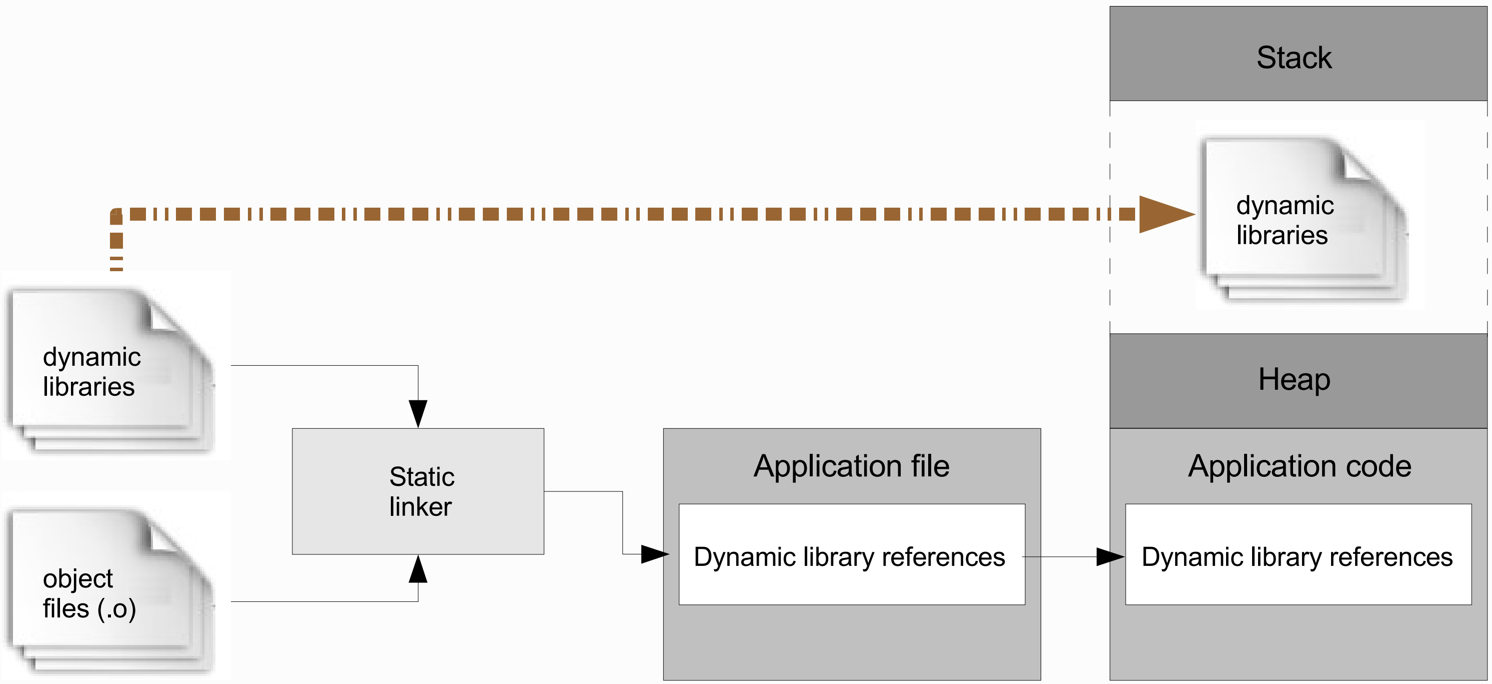
Static linking (no system dependencies)



Static linking (no system dependencies)

- Executable modules are rarely totally static
  - User's own libraries are usually static
  - System libraries are usually dynamic (except on some platforms like SUPER-UX where shared objects do not exist)
  - Application libraries (commercial as well as open source) are often dynamic

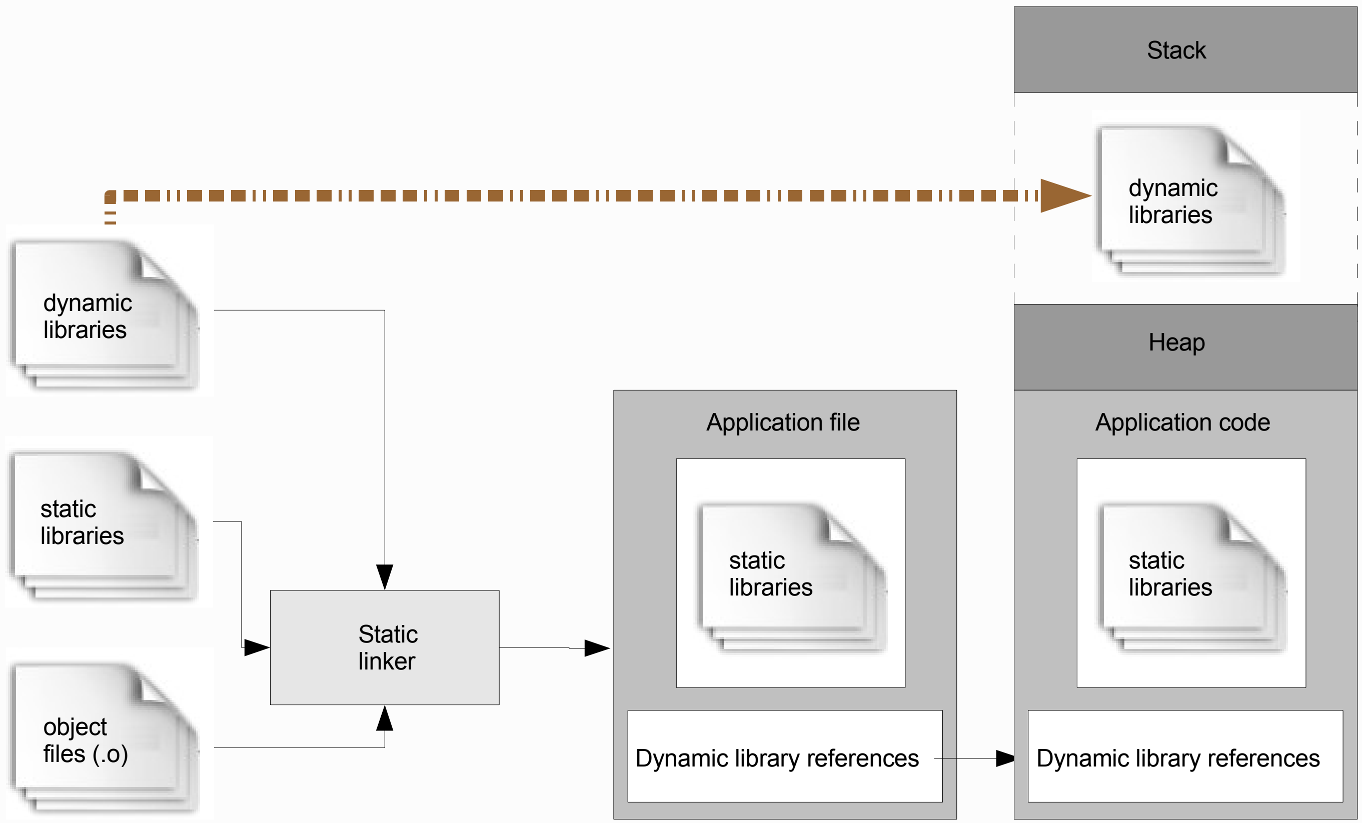
Dynamic linking (simple case)



Static + dynamic linking (most usual case)

- The normal state of affairs is a mix of static and dynamic libraries

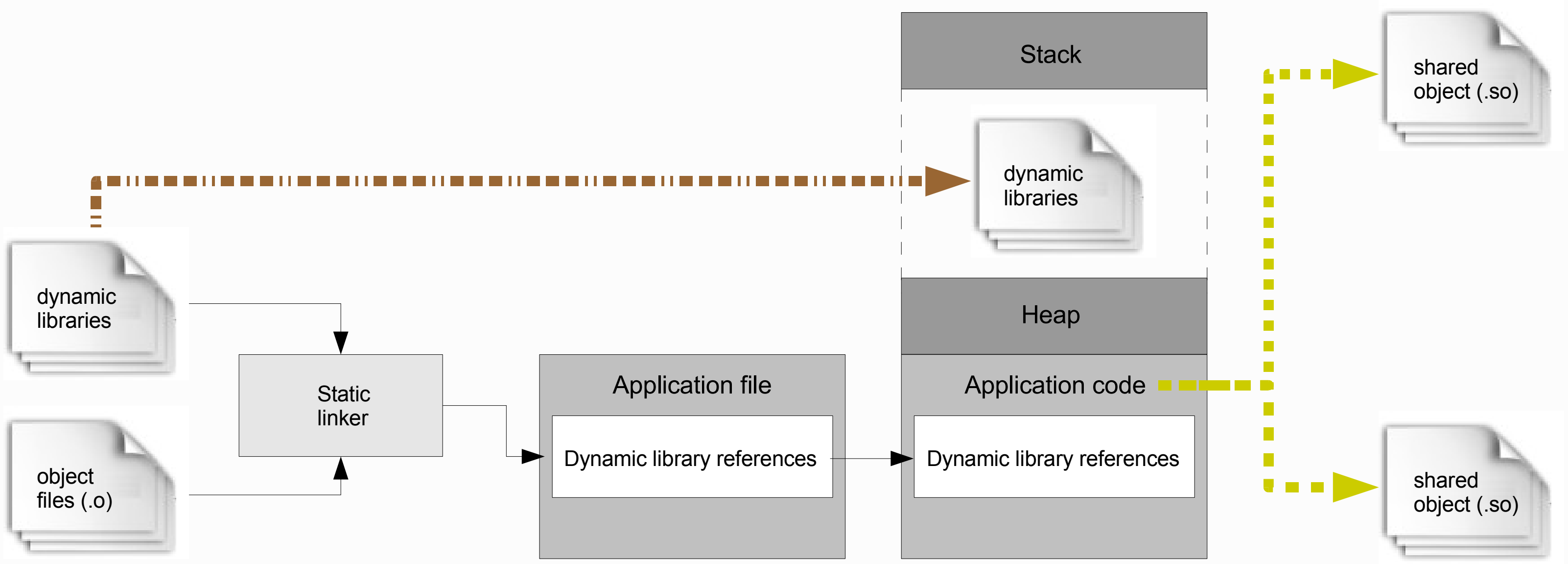
Static + dynamic linking (most usual case)



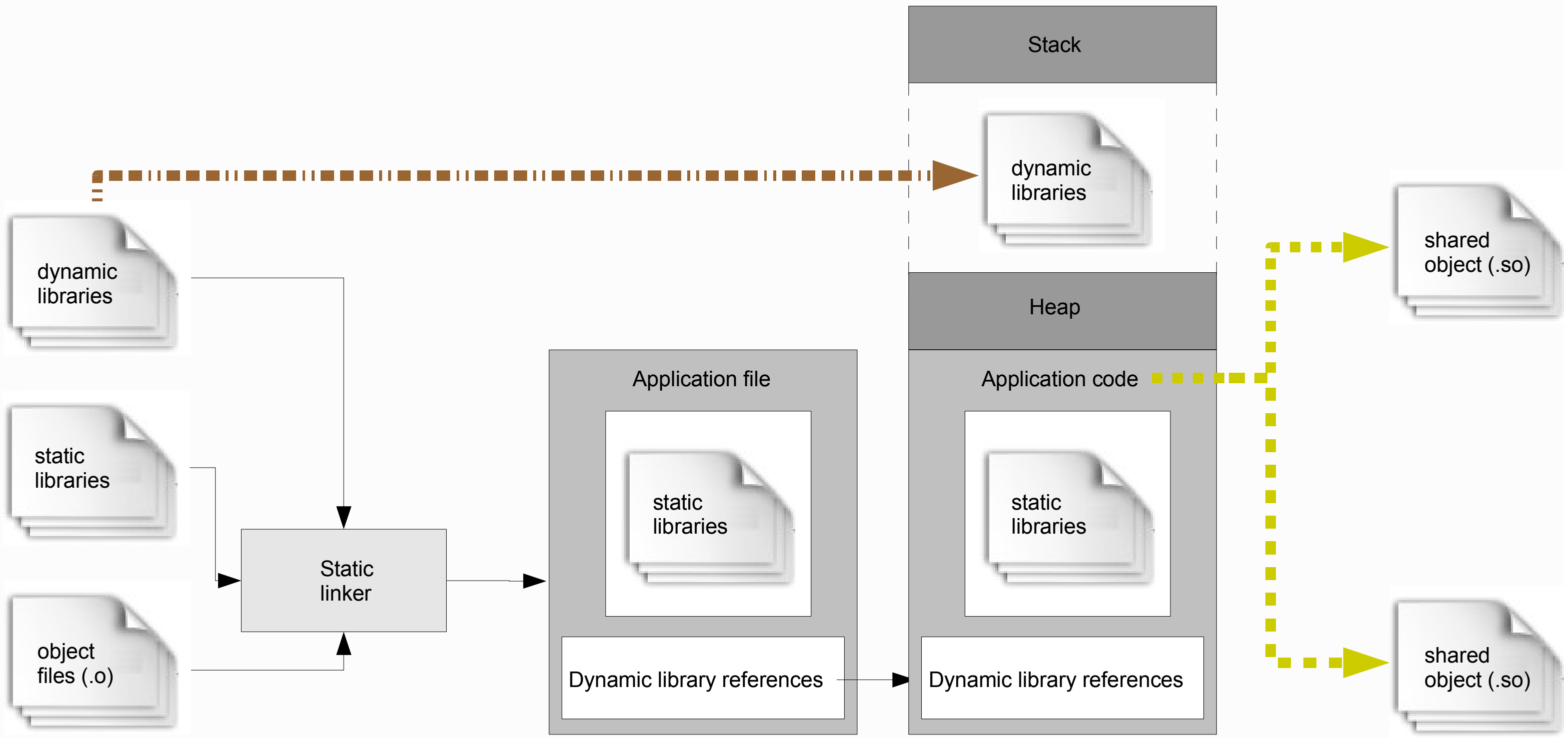
Runtime linking

- Run time dynamic linking is also used by software where the name of the modules to be called is not known in advance but is determined at run time (Python, Perl, Matlab, IDL, ...)

Runtime linking

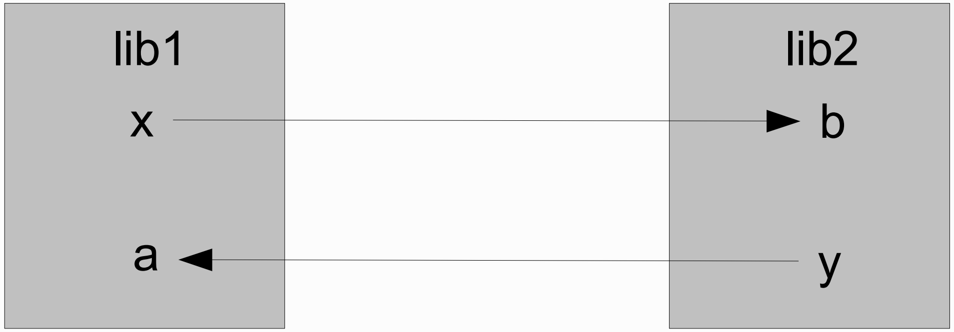


The whole show



Predictability of results

- Results are not always as intuitive as they seem
  - Library order
    - Especially when a symbol is defined in more than one library
  - When is a symbol dependency noticed ?
  - On pass linker side effects
    - x from lib1 calls b from lib2 and y from lib2 calls a from lib1
  - Some linkers are smarter than others (and therefore produce different results)



Which library will I end up using ?

- Static libraries
  – Produce (and read) a load map
- Dynamic libraries
  – ldd will tell

Which library will I end up using ?

A practical example

Library libx.a contains

```
subroutine a ! suba1.f90
print *,'this is a from library x'
return
end
subroutine b ! subb.f90
print *,'this is b from library x'
return
end
```

Library liby.a contains

```
subroutine a ! suba2.f90
print *,'this is a from library y'
return
end
subroutine c ! subc.f90
print *,'this is c from library y'
call a
return
end
```

```
Program test1
call b
call c
stop
end
```

```
Program test2
call a
call b
call c
stop
end
```

Which library will I end up using ?

- Created with
- FC -c sub*.f90
- ar rcv libx.a suba1.o subb.o
- ar rcv liby.a suba2.o subc.o
- FC test1.f90 -L. -lx -ly -o test1
- FC test2.f90 -L. -lx -ly -o test2
- FC test2.f90 subc.f90 -L. -lx -ly -o test3
- (FC is the appropriate compiler name)
- What will the execution output of test1 and test2 be

Which library will I end up using ?

- IBM AIX output
  - c6f14p4m 5% ./test1
  - this is b from library x
  - this is c from library y
  - this is a from library x
  - c6f14p4m 6% ./test2
  - this is a from library x
  - this is b from library x
  - this is c from library y
  - this is a from library x
  - c6f14p4m 7% ./test3
  - this is a from library x
  - this is b from library x
  - this is c from library y
  - this is a from library x
- Linux output
  - averroes 509% ./test1
  - this is b from library x
  - this is c from library y
  - this is a from library y
  - averroes 510% ./test2
  - this is a from library x
  - this is b from library x
  - this is c from library y
  - this is a from library x
  - averroes 511% ./test3
  - this is a from library x
  - this is b from library x
  - this is c from library y
  - this is a from library x

Which library will I end up using ?

- Why did we get a difference
  - Linux uses a one pass no look back gnu linker
  - test1 calls b and c
  - The loader goes through libx looking for b and c
  - The loader finds b, uses it, does not find c
  - The loader then goes through liby, finds c, uses it, discovers that c needs a.
  - The loader DOES NOT LOOK BACK into libx, finds a in liby uses it from libb
- What if suba2.o is removed from liby on linux?
  - r.f90 test1.f90 -L. -lx -ly -o test1
  - ./liby.a(subc.o): In function `c_':
  - subc.f90:(.text+0xbe): undefined reference to `a_'

Which library will I end up using ?

- Different outcome on AIX
  - The AIX linker remembers the order of the libraries
  - test1 calls b and c
  - The loader goes through libx looking for b and c
  - The loader finds b, uses it, does not find c
  - The loader then goes through liby, finds c, uses it, discovers that c needs a.
  - The loader LOOKS BACK into libx, finds a in libx, uses it from libx
- What if suba2.o is removed from liby on AIX ?
  - xlf test1.f90 -L. -lx -ly -o test1
  - ** test === End of Compilation 1 ===
  - 1501-510 Compilation successful for file test.f90.

Which library will I end up using ?

- What happened for test3
  - The linker loads test3.o and subc.o
  - test3 calls b and c, c calls a
  - The linker already has c from subc.o
  - The loader goes through libx looking for b and a
  - The loader finds b and a in liby, uses them
  - The loader then goes through liby, needs nothing any more.

Common block init problem (1)

```
averroes 525% cat prog1.f90
program prog1
call sub01
call sub02
stop
end
```

```
averroes 526% cat
sub01.f90
subroutine sub01
common /my_common/a,b
data a / 1.0/
print *,'subroutine sub01'
return
end
```

```
averroes 527% cat sub02.f90
subroutine sub02
common /my_common/a,b
data b / 2.0/
print *,'subroutine sub02'
return
end
```

```
averroes 534% cat sub03.f90
subroutine sub02
common /my_common/a,b
print *,'subroutine sub02'
return
end
```

FC prog1.f90 sub01.f90 sub02.f90

```
prog1.f90:
sub01.f90:
sub02.f90:
sub02.o: In function `.C1_302':
sub02.f90:(.data+0x30): multiple definition of `my_common_'
sub01.o:sub01.f90:(.data+0x30): first defined here
```

FC prog1.f90 sub01.f90 sub03.f90

```
prog1.f90:
sub01.f90:
sub02.f90:
```

Common block init problem (2)

```
averroes 530% nm sub01.o
00000018 d .C1_283
00000010 d .C1_285
00000020 d .C1_300
0000002c d .C1_302
0000001c d .C1_303
00000000 d .C1_306
00000014 d .C1_308
         U _GLOBAL_OFFSET_TABLE_
000000c0 t __sub01_END
00000030 D my_common_
         U pgf90_compiled
         U pgf90io_ldw
         U pgf90io_ldw_end
         U pgf90io_ldw_init
         U pgf90io_src_info
00000010 C pghpf_0_
00000001 C pghpf_0c_
00000020 C pghpf_0l_
00000004 C pghpf_lineno_
00000004 C pghpf_me_
00000004 C pghpf_np_
00000010 T sub01_
```

```
averroes 531% nm sub02.o
00000018 d .C1_283
00000010 d .C1_285
00000020 d .C1_300
0000002c d .C1_302
0000001c d .C1_303
00000000 d .C1_306
00000014 d .C1_308
         U _GLOBAL_OFFSET_TABLE_
000000c0 t __sub02_END
00000030 D my_common_
         U pgf90_compiled
         U pgf90io_ldw
         U pgf90io_ldw_end
         U pgf90io_ldw_init
         U pgf90io_src_info
00000010 C pghpf_0_
00000001 C pghpf_0c_
00000020 C pghpf_0l_
00000004 C pghpf_lineno_
00000004 C pghpf_me_
00000004 C pghpf_np_
00000010 T sub02_
```

Common block init problem (3)

```
averroes 531% nm sub02.o
00000018 d .C1_283
00000010 d .C1_285
00000020 d .C1_300
0000002c d .C1_302
0000001c d .C1_303
00000000 d .C1_306
00000014 d .C1_308
         U _GLOBAL_OFFSET_TABLE_
000000c0 t __sub02_END
00000030 D my_common_
         U pgf90_compiled
         U pgf90io_ldw
         U pgf90io_ldw_end
         U pgf90io_ldw_init
         U pgf90io_src_info
00000010 C pghpf_0_
00000001 C pghpf_0c_
00000020 C pghpf_0l_
00000004 C pghpf_lineno_
00000004 C pghpf_me_
00000004 C pghpf_np_
00000010 T sub02_
```

```
averroes 536% nm sub03.o
00000018 d .C1_283
00000010 d .C1_285
00000020 d .C1_300
0000002c d .C1_302
0000001c d .C1_303
00000000 d .C1_306
00000014 d .C1_308
         U _GLOBAL_OFFSET_TABLE_
000000c0 t __sub02_END
00000008 C my_common_
         U pgf90_compiled
         U pgf90io_ldw
         U pgf90io_ldw_end
         U pgf90io_ldw_init
         U pgf90io_src_info
00000010 C pghpf_0_
00000001 C pghpf_0c_
00000020 C pghpf_0l_
00000004 C pghpf_lineno_
00000004 C pghpf_me_
00000004 C pghpf_np_
00000010 T sub02_
```

Common block init problem (4)

- FC prog1.f90 sub01.f90 sub02.f90
  - Initializing a common block with a data statement DEFINES a symbol with the name of the common block
  - sub01 defines my_common
  - sub02 defines my_common
  - OOPS, duplicate symbol
- FC prog1.f90 sub01.f90 sub03.f90
  - Declaring a common block only produces a REFERENCE to en elsewhere defined symbol
  - sub01 defines my_common
  - sub02 references my_common
  - Everybody is happy
- Common blocks must be initialized in ONE PLACE only

Controlling dynamic libraries (AIX)

- Side effects of switching compiler libraries (1)
  - xlf test1.f90 -o test1 -L. -lx -ly
  - LIBPATH= xlf=/usr/bin/xlf
  - test1 needs:
  - /usr/lib/libc.a(shr.o)
  - /usr/lpp/xlf/lib/libxlf90.a(io.o)
  - /unix
  - /usr/lib/libcrypt.a(shr.o)
  - LIBPATH=/opt/ssm/XLF_12.1.0.0_aix53-ppc-64/usr/lib
  - test1 needs:
  - /usr/lib/libc.a(shr.o)
  - /opt/ssm/XLF_12.1.0.0_aix53-ppc-64/usr/lib/libxlf90.a(io.o)
  - /unix
  - /usr/lib/libcrypt.a(shr.o)

Controlling dynamic libraries (AIX)

- Side effects of switching compiler libraries (2)
  - LIBPATH=/opt/ssm/XLF_12.1.0.0_aix53-ppc-64/usr/lib:
    xlf=/opt/ssm/XLF_12.1.0.0_aix53-ppc-64/bin/xlf
  - xlf test1.f90 -o test1b -L. -lx -ly
  - test1b needs:
  - /usr/lib/libc.a(shr.o)
  - /opt/ssm/XLF_12.1.0.0_aix53-ppc-64/usr/lib/libxlf90.a(io.o)
  - /unix
  - /usr/lib/libcrypt.a(shr.o)
  - forcing LIBPATH=
  - test1b needs:
  - /usr/lib/libc.a(shr.o)
  - /opt/ssm/XLF_12.1.0.0_aix53-ppc-64/usr/lib/libxlf90.a(io.o)
  - /unix
  - /usr/lib/libcrypt.a(shr.o)

Controlling dynamic libraries (AIX)

- What happened ?
- Case (1)
  - We were using the DEFAULT compiler and set of libraries
  - No special information left in test1 executable
  - LIBPATH forces usage of dynamic libraries from that path
- Case (2)
  - We were using a special instance of the compiler, that also defined the LIBPATH environment variable
  - Information about dynamic libraries to be used was left INSIDE the executable
  - LIBPATH forces usage of libraries from that path
  - If LIBPATH is absent the path to the proper dynamic libraries is taken from the executable

Controlling dynamic libraries (AIX)

- How to play it safe on AIX
  - When creating executables
    - Suppress default compiler
    - Explicitly use a compiler version (. r.ssmuse.dot ....)
  - In jobs that run these executables
    - Suppress default compiler (do not risk forcing another version)
    - Use ldd the check where the libraries come from
- Surprise scenario
  - Create executable on maia or saiph using system defaults
  - Run executable on other machine using system defaults
  - Get different results (#@!$%)
  - This can happen because the executable will look for the libraries in the default place and not find the same version

Controlling dynamic libraries (Linux)

- Same method as under AIX
  - LD_LIBRARY_PATH is used instead of LIBPATH (so much for standards between members of the *NIX family)
- With an extra twist in the plot
  - Our local default setup for the Portland Group compiler forces most FORTRAN runtime libraries to the STATIC version libraries
  - Exception: programs using OpenMP need at least one dynamic library from the PGI runtime (libnuma.so)
- Other linux compilers
  - Gfortran (etch systems only) is fully dynamic
  - SunStudio compiler setup also uses the STATIC libraries except for OpenMP

A few more tricks (AIX/Linux)

- I have a .o file, which compiler was used to generate it ? (AIX xlf/xlc; Linux PGI, SunStudio, gfortran,gcc)
  - strings -a xxx.o | grep '\(PGF90\)\|\(Sun Fort\)\|\(GCC:\)\|\(IBM XL\)' (for an object file)
  - strings -a yyy.a | grep '\(PGF90\)\|\(Sun Fort\)\|\(GCC:\)\|\(IBM XL\)' (for a library)
- I have a .o file, what external symbols does it need
  - nm xxx.o | grep ' U '
- Does library libz.a define symbol xyz
  - nm libz.a | grep xyz | grep ' T '

Compiling and testing at CMC

- Where do i edit and manage sources ?
  - MY workstation (sources should be on /home/..., /home filesystems are mounted an ALL machines)
- Where do I compile ?
  - MY workstation (zeta head node if IBM application) (yes, the very same linux compilers are available on all workstations and servers)
  - For delivery to operations : erg (saiph/zeta if AIX application)
  - If i do not have a workstation: erg
- Where do i test ?
  - MY workstation (interactive or batch)
  - alef development cluster (BATCH ONLY)
  - If i do not have a workstation: erg
  - zeta/saiph (light interactive or BATCH)

Compilers

- native FORTRAN compilers (main)
  - AIX:
    - IBM xlf (versions 10.1, 11.1, 12.1)
      - . s.ssmuse.dot Xlf10 (64)
      - . s.ssmuse.dot Xlf11 (64)
      - . s.ssmuse.dot Xlf12 (64 , recommended)
  - Linux: (32 bit version etch, 64 bit version kubuntu 9.10)
    - PGI pgf90 (versions 6.1, 6.2, 7.2, 8.0, 9.0, 10)
      - . s.ssmuse.dot pgi6 (32, legacy, not recommended)
      - . s.ssmuse.dot pgi6xx (32, legacy)
      - . s.ssmuse.dot pgi7xx (32 , not recommended)
      - . s.ssmuse.dot pgi8xx (32 , not recommended)
      - . s.ssmuse.dot pgi9xx (32/64, recommended , CUDA 64 bit)
      - . s.ssmuse.dot pgi10xx (32/64, stable , CUDA 64 bit)
      - . s.ssmuse.dot pgi11xx (64, bleeding edge , CUDA 64 bit)

Compilers

- native FORTRAN compilers (others)
  - AIX:
    - NAG f95 (r.nagf95)
  - Linux:
    - SUN sunf90
    - GNU gfortran
    - NAG f95 (r.nagf95)
- native C compilers
  - AIX: IBM xlc
  - Linux: GNU cc/gcc, SUN suncc

The local tools (meta compilers)

- multi platform compilation tools
  (calling sequence similar to native compilers, forces library and I/O compatibility options, can be used with ./configure ; make install)
  also relies on Compiler_rules configuration file
  - s.f77, s.f90, s.cc, s.CC, s.ftn, s.ftn90, s.GPPF, s.GPPF90
- multi language "universal" compiler
  (calling sequence NOT compatible with native compilers)
  - s.compile
- Configuration controlled
  - EC_ARCH / BASE_ARCH / COMP_ARCH environment variables
    ex: COMP_ARCH=pgi9xx, BASE_ARCH=Linux, EC_ARCH=Linux/pgi9xx
  - Compiler_rules files

Implicit paths

- Compile time
  - Automatically added to include search path
    - $EC_INCLUDE_PATH
    - $ARMNLIB/include
    - $ARMNLIB/include/$EC_ARCH
- Build time
  - Automatically added to library search path
    - $EC_LD_LIBRARY_PATH

The local tools (extra arguments)

- Disassembled / reassembled options
  - `-O n (different from -On)`
  - `-Dname=value`
  - `-Iinclude/path`
  - `-lmy_lib`
  - `-Llibrary/path`
  - `-o output_file`

The local tools (extra arguments)

- Extras (some borrowed from s.compile)
  - `-verbose`
  - `-openmp`
  - `-mpi`
  - `-src source_file`
  - `-debug`
  - `-shared`
  - `-dynamic`
  - `-prof`

s.compile

- Multi language support using recognized extensions
  - FORTRAN with custom preprocessing: .ftn, .ptn, .ftn90, .cdk90
  - FORTRAN with own preprocessing: .F, .F90
  - FORTRAN: .for, .f, .f90
  - C: .c
  - C++: .cc, .cpp, .cxx, .c++, .C, .CPP, .cp
  - assembler: .s
- Driven by a configuration file (Compiler_rules)
  - User may override with own configuration
    $HOME/userlibs/$EC_ARCH/Compiler_rules
  - Forces library and I/O compatibility options
  - Multiple architecture support (EC_ARCH env variable)

s.compile arguments (1)

- -src (sources, .f, .F, .for, .ftn, .ptn, .f90, .F90, .cdk90, .c, .C, .CPP, .cxx, .c++, .cc, .cp, .s)
- -O (optimization, 0-4)
- -openmp
- -mpi
- -debug / -prof (use debugger / profiler)
- -includes (path for includes files, #include and include)
- -defines (=-Dname1=value2:-Dname2=value2 ...)
- -P (preprocessor pass only)
- -opt / -optc / -optf (=-option1:-option2:-option3 ...)
  compiler options / C options / Fortran options

s.compile arguments (2)

- -obj (objects to add to the executable)
- -codebeta (object modules to add from environment)
- -libpath (path to be searched for libraries)
- -libappl (application libraries to use)
- -librmn (rmnlib version rmn_010 / rmnbeta_9.0 ...)
- -libsys (system libraries to use)
- -o (name of the executable)
- -shared (build a shared object [partial support])
- -conly (use cc to build executable instead of Fortran)

s.compile examples

- Fortran code

```
s.compile -src tdgauss.f90 -o a.out -librmn rmnbeta_011
s.compile -src tdgauss.f90 -o a.out -librmn rmn_010
```

- C main, no Fortran library involved

```
s.compile -o a.out -src r.abs_to_rel.c -conly
```

- C main , renamed, using rmnlib, built with Fortran

```
s.compile -src cmain.c -bidon c -o a.out -librmn rmn_010 \
            -defines=-Dmain=mymain -main mymain
```

- OpenMP Fortran program

```
s.compile -o a.out -openmp -src simu_opt30_nodata4.f -O 3
```

- MPI + OpenMP

```
s.compile -o a.out -openmp -src simu_opt30_nodata_mpi.f -mpi \
            -O 3 -libappl rpn_comm301 -librmn rmn_010
```

Topics to add ?

- Send suggestions to
  - service.hpcs@ec.gc.ca

The END

Thank you for your attention