Lecture 04 – Elements of Development: *Configuring, Compiling, Linking*

Prof. Brendan Kochunas

NERS/ENGR 570 - Methods and Practice of Scientific Computing (F20)

COLLEGE OF ENGINEERING
NUCLEAR ENGINEERING & RADIOLOGICAL SCIENCES
UNIVERSITY OF MICHIGAN

Outline

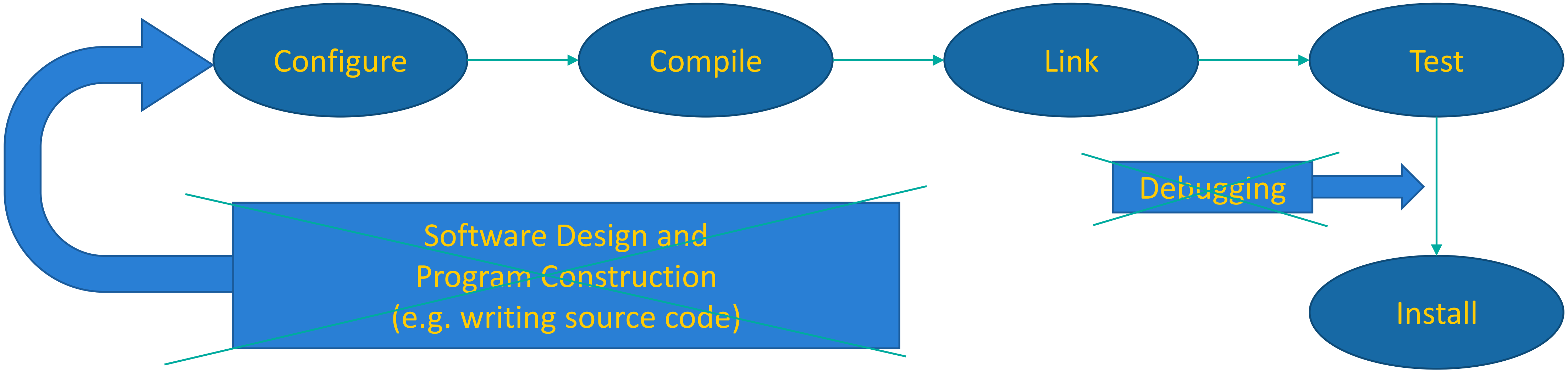
- Introduction
- Configuring
- Compiling
- Linking
- Program Design, Infrastructure, Testing, Debugging will be covered in other lectures ☺

Learning Objectives: By the end of Today's Lecture you should be able to

- (*Knowledge*) explain what happens at each step of configuring, compiling, and linking
- (Skill) How to troubleshoot compilation
- (Skill) How to troubleshoot linking

Elements to Development

- Program Design, Testing, and Debugging are all integral parts of development.
  - They each will get their own lecture later in the semester.
- This lecture is focused on the tools and steps needed to build software:



The “Toolchain”

Configure → Compile → Link → Test → Install

Definitions

- The tools (e.g. programs and their libraries) typically used by the developer that are needed to take your source files (as input) and produce executables.
- Frequently the definition of a tool chain will also include:
  - a program for “configuring”
  - a “make” script and program
  - a distribution of MPI
- Third Party Libraries (TPLs) are typically not considered part of the “toolchain”
  - MPI is sort of the exception to this
- Toolchains represent part of the “Minimum Requirements” for a software package.
  - Software packages can also support multiple toolchains

Examples

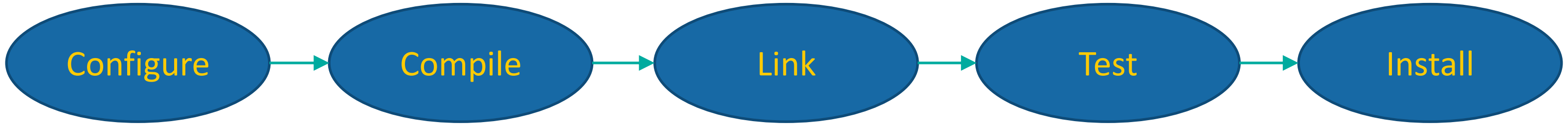
- In the CASL project
  - CMake 2.8.11
  - autoconf-2.69 (auto tools)
  - gnu-4.8.3 compilers
  - mpich-3.1.3 (MPI library)
  - Assumes GNU Make or Make is available on the system
    - Almost always available on linux since it may be needed to install (parts of) the OS
- Other examples
  - Trilinos – supports many
  - PETSc – supports many
  - xSDK – being defined for future computational science and engineering applications

Some Terminology about "Time"

- *configure-time* – when you are configuring
- *compile-time/build-time* – when you are compiling
- *link-time* – when you are linking (very last step in "compiling")
- *run-time* – when you are running the executable
- Important for communicating "when things went wrong"

An Imperfect Cooking Analogy

Any volunteers that do not mind sharing details of their kitchen?



Configuring: Preparing to cook

Cooking

- Look around and find all the necessary ingredients and cookware (pots, pans, knives) to make dinner
    - Your stomach
    - Appliances
    - Spatula, spoon and knives
    - Salt, Pepper, Spices and Sauces
- Detailed instructions to cook in your kitchen
- Cookbooks
- Writing your own cookbook for others
- It's a part of life
- Use pre-packaged frozen meals

Software

- Purpose is to probe the system for to determine
    - the computer architecture (e.g. x86, AIX, ARM, etc.)
    - what compilers are installed, and where they are installed
    - what additional system software is installed
    - what third party libraries are available
- Basically creates "Makefiles" for a specific computer and environment for use in compiling (the next step)
- Specialized programs exist to perform the "configure" step.
    - autotools (autoconf/automake) → `configure`
    - CMake → `cmake`
- Most difficult part to establishing complex software systems.
- Considered part software infrastructure
- AUTOMATE as much as possible

Compiling: Cooking

Cooking

- Basically following the steps
  - Chop the onion
  - Season the ....
  - Fry the egg
  - Bake in the oven
- Lots of different kinds of steps/things we do in cooking
- Objective: prepare everything in the meal to go on the plate

Software

- Compilers do lots of different things.
  - Preprocessing
  - Optimizing
  - Check for
  - Give warnings
  - Generate assembly
  - Include this other file
- Objective: prepare all the machine code to go into the executable

Linking: Plating the Meal

Cooking

- Objective: put everything you've made together into a delicious meal.
- Considerations
  - Where did I put this cooked thing
  - Fresh parmesan?
  - Bowl or Plate
  - Spoon or Fork

Software

- Objective: put everything together into a functional executable
- Considerations
  - Where is this function/library coming from?
  - Library or executable
  - Shared or static

Configuring Software

Things that Configuring Can Control

- Where the libraries and executables produced from compilation are installed
- Compiler options:
  - e.g. whether the libraries and executables are “debug” (slow) or “release” (fast)
  - e.g. whether the compilation produces “dynamic” or “static” binaries
- What features of the library are enabled
  - e.g. with HDF5 you can specify whether you want the compiled library to include Fortran interfaces or just C interfaces.
  - e.g. what third party libraries to include (often provide additional capability in the software package
    - In HDF5 the “Z” library can be included to provide data compression.
- Various other options that would be specific to the software package.

Configuration Options

The Cliff's Notes

- Depends on the tool!
- You will *always have to read documentation*
  - Usually files are named `INSTALL` or `README`
- Hopefully libraries and programs you work with are well documented
- Make sure you document your configuration steps well (or make them "robust") in software you produce
  - People won't use your software if it is difficult to install or build.

"Common" Usage and Options

- Autotools (autoconf/automake)
  - `./configure [options]`
    - `--prefix=<path_to_install>`
    - `CC=<c_compiler>`
    - `FC=<fortran_compiler>`
- CMake
  - `cmake [options] <path_source_dir>`
    - `-DCMAKE_BUILD_TYPE="Release"`
    - `-DCMAKE_C_COMPILER`
    - `-DCMAKE_CXX_COMPILER`
    - `-DCMAKE_Fortran_COMPILER`

Examples of Configuration Scripts

CMake (CMakeLists.txt) for METIS

```
cmake_minimum_required(VERSION 2.8)
project(METIS)

set(GKLIB_PATH "GKlib" CACHE PATH "path to GKlib")
set(SHARED FALSE CACHE BOOL "build a shared library")
if(MSVC)
  set(METIS_INSTALL FALSE)
else()
  set(METIS_INSTALL TRUE)
endif()
# Configure libmetis library.
if(SHARED)
  set(METIS_LIBRARY_TYPE SHARED)
else()
  set(METIS_LIBRARY_TYPE STATIC)
endif(SHARED)
include(${GKLIB_PATH}/GKlibSystem.cmake)
# Add include directories.
include_directories(${GKLIB_PATH})
# Recursively look for CMakeLists.txt in subdirs.
add_subdirectory("include")
add_subdirectory("libmetis")
add_subdirectory("programs")
```

Autoconf (configure.in) for PAPI

```
AC_PREREQ(2.59)
AC_INIT(PAPI, 5.5.0.0, ptools-perfapi@eecs.utk.edu)
AC_CONFIG_SRCDIR([papi.c])
AC_CONFIG_HEADER([config.h])

AC_MSG_CHECKING(for architecture)
AC_ARG_WITH(arch,
                 [  --with-arch=<arch>    Specify architecture (uname -m)],
                 [arch=$withval],
                 [arch=`uname -m`])
AC_MSG_RESULT($arch)

AC_ARG_WITH(bitmode,
             [  --with-bitmode=<32,64>  Specify bit mode of library],
             [bitmode=$withval])

AC_MSG_CHECKING(for OS)
```

Autotools Configure Example Cont.

```
#!/bin/sh
# Guess values for system-dependent variables and create Makefiles.
# Generated by GNU Autoconf 2.59 for PAPI 5.5.0.0.

# Identity of this package.
PACKAGE_NAME='PAPI'
PACKAGE_TARNAME='papi'
PACKAGE_VERSION='5.5.0.0'
PACKAGE_STRING='PAPI 5.5.0.0'
PACKAGE_BUGREPORT='ptools-perfapi@eecs.utk.edu'

ac_unique_file="papi.c"
# Factoring default headers for most tests.
ac_includes_default="\
#...
#if STDC_HEADERS
# include <stdlib.h>
# include <stddef.h>
#else
# if HAVE_STDLIB_H
#  include <stdlib.h>
# endif
#endif
```

- As a user you typically will not run `autoconf`
  - But as a developer you might
- As a user you will run configure shell script produced by autoconf.
- Conventional wisdom for developers
  - CMake >> ~~Autohell~~ Autotools
- **If starting new, start with CMake.**

Troubleshooting the Configure Step

<u>For Scientific Software Development</u>

- My configure failed! What do I do?
  - Uh oh… Usually difficult to resolve
- At best, problem is solved by modifying your environment.
- At worst, may require some other software be installed
  - And you may not have privileges to do so!
  - In this case you're likely 
  - …unless you're willing to put in way more time than is appropriate

<u>For Cooking</u>

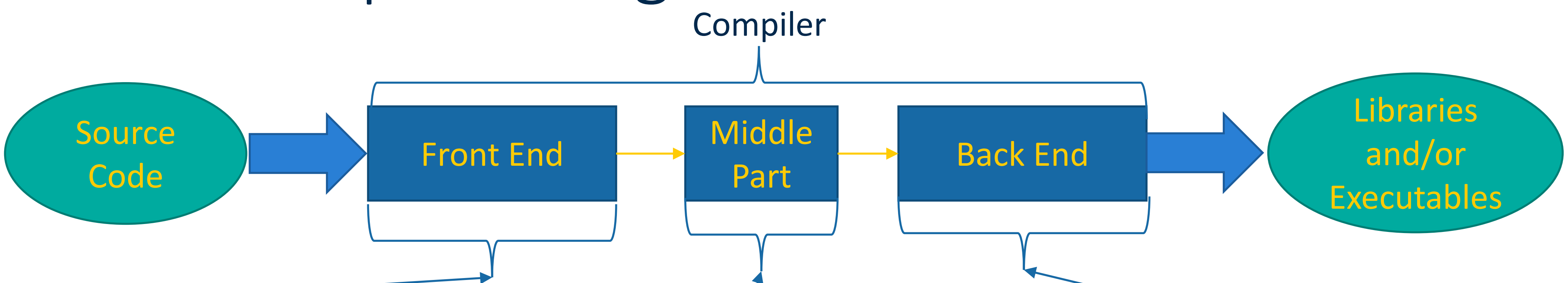
- I can't find everything I need to cook dinner!
  - I'm hungry now, but I need to grill this steak!
- Ask your neighbor for a cup of sugar and some eggs or a melon baller
- You mean I need a grill!?
  - But I live in an apartment!
  - Guess you'll go hungry
  - …unless I can make friends with someone who has a grill

Compilers/Compiling

What is a Compiler?

- Translates a high-level programming language (suitable for humans) into a low level machine language required by the computer.
- Typically have several common features
  - Checking for syntax and programming errors
  - Supply debugging information
  - Perform optimizations
- It’s a program written by people
  - so it has bugs
  - And different versions and the behavior between versions can vary significantly.

Modern Compiler Program Architectures



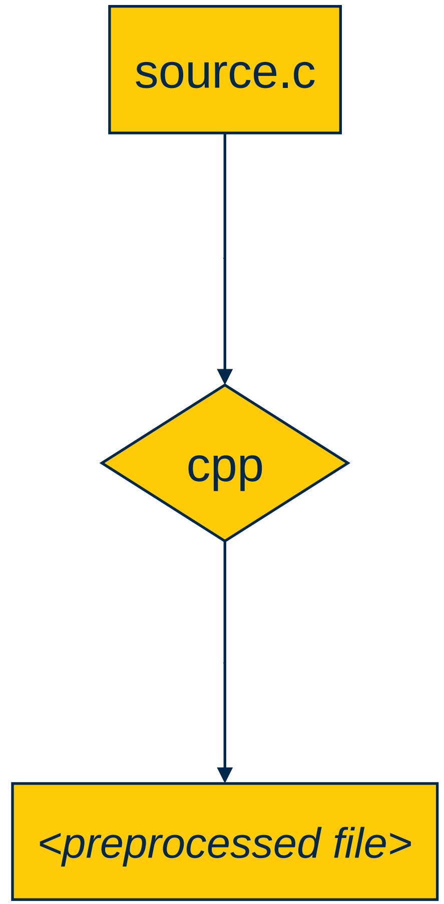
- Lexical Analysis
  - Read and parse text in source files into *tokens*
- Syntax Analysis
  - Arrange tokens in syntax tree to reflect program structure
- Type checking
  - Checks syntax tree for mistakes (e.g. undefined variables)

- Intermediate coded generation
  - Translation to simple machine independent language
  - Typically will vary between compilers
- Optimization
  - Apply algorithms for optimization to intermediate language

- Register allocation
  - Translate variables to machine registers (memory locations)
- Machine code generation
  - Translate intermediate language to machine code (assembly)
- Assembly and linking
  - Convert assembly to binary and resolve addresses for variables and functions

Other things a compiler does (sort of)

- Preprocess (separate program executed by compiler during compilation, "**cpp**")
  - ***Modifies source files***
  - A part of C, but can be used in the compilation of C++ & Fortran
    - In Fortran file extensions control default behavior:
    - *.F, *.F90 are automatically preprocessed, and *.f, *.f90 are not
  - Based on "directives", start with "#" in first column.
    - Include files, macro expansion, conditional compilation
  - Compilers will predefine some symbols for you
- Link (separate program executed by compiler during compilation, "**ld**")
  - We'll discuss linking in a few slides…



Some Preprocessor Examples

C

```c
#include <stdio.h>

//Define macros
#define PI 3.14159
#define RADTODEG(x) ((x) * 57.29578)

int main()
{
    float x;
    x=RADTODEG(PI*0.5);
    printf("PI/2 radians in degrees is %.6f\n",x);
    return 0;
}
```

```c
int main()
{
    float x;
    x=((3.14159*0.5) * 57.29578);
    printf("PI/2 radians in degrees is %.6f\n",x);
    return 0;
}
```

Fortran

```fortran
PROGRAM main

#ifdef __GFORTRAN__
  WRITE(0,'(a,i2,a)') 'File: "'//__FILE__//'", line ',__LINE__, &
    " was compiled with gfortran!"
#else
  WRITE(0,'(a,i2,a)') 'File: "'//__FILE__//'", line ',__LINE__, &
    " was NOT compiled with gfortran!"
#endif

ENDPROGRAM
```

```fortran
PROGRAM main


  WRITE(0,'(a,i2,a)') 'File: "'//"hello.F90"//'", line ',4, &
    " was compiled with gfortran!"


ENDPROGRAM
```

Compiler options for debugging

GCC compiler option	Meaning
`-g`	Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.
`-fsanitize=<opt>`	Enable AddressSanitizer, a fast memory error detector.
`-fbounds-check`	Generate additional code to check that indices used to access arrays are within the declared range during run time.
`-fcheck-pointer-bounds`	Each memory reference is instrumented with checks of the pointer used for memory access against bounds associated with that pointer.
`-fstack-check`	Generate code to verify that you do not go beyond the boundary of the stack.

Pretty much only `-g` is important.
For the other run-time checks, significant overhead in run time may be observed.

Other compiler options

GCC compiler option	Meaning
`-c`	Compile without linking
`-o`	Name of output file.
`-I`	Search path for included header files (there are predefined system paths)
`-L`	Search path for libraries
`-l`	Library name to link in
`-D` ***`<symbol>`***	Define preprocessor symbol ***`<symbol>`*** during compilation
`-E`	Output preprocessed source file
`-S`	Output assembly from compilation
`-fPIC`	Compile **P**osition **I**ndependent **C**ode (necessary for shared objects)
`-fopenmp`	Process OpenMP directives
`-p`	Generate profiling information during run time for profiling analysis tools (e.g. `gprof`)
`-ftest-coverage`	Generate coverage information during run time for coverage analysis tools (e.g. `gcov`)

Object code & Binary output

- Compiling a source file `source.F90` (e.g. –c) produces an object file `source.o`
  - Object files are *relocatable* machine code.
  - Typical object file format for linux is ELF.
  - Cannot view object files in text editors
- ELF files contain
  - Program header table describing 0 or more segments
    - Contains run-time information
  - Section header table describing 0 or more sections
    - Contains link-time information
  - Data referred to by segments and sections
- How can you inspect ELF files/object code/object files?

Snippet of object file from program on slide 15 in vi

```
^?ELF^B^A^A^@^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@¸^B^@^@^@^@^@^@^
@^@^@^@@^@^@^@^@^@@^@^M^@
^@UH<89>åH<81>ìà^A^@^@HÇ<85>(þÿÿ^@^@^@^@Ç<85>0
þÿÿ^E^@^@^@HÇ<85>hþÿÿ^@^@^@^@Ç<85>pþÿÿ^H^@^@^@
Ç<85> þÿÿ^@^P^@^@Ç<85>$þÿÿ^@^@^@^@H<8d><85>
þÿÿH<89>Çè^@^@^@^@H<8d><85>
þÿÿ°^X^@^@^@¾^@^@^@^@H<89>Çè^@^@^@^@H<8d><85>
þÿÿ°^D^@^@^@¾^@^@^@^@H<89>Çè^@^@^@^@H<8d><85>
þÿÿ°^\^@^@^@¾^@^@^@^@H<89>Çè^@^@^@^@H<8d><85>
þÿÿH<89>Çè^@^@^@^@ÉÃUH<89>åH<83>ì^P<89>}üH<89>
uðH<8b>Uð<8b>EüH<89>Ö<89>Çè^@^@^@^@¾^@^@^@^@¿^
H^@^@^@è^@^@^@^@è^Xÿÿÿ¸^@^@^@^@ÉÃ^@^@^@^@^@^@^
@^@^@^@^@^@^@^@^@^@hello.F90^@(a,i2,a)File:
"hello.F90", line ^@^@^D^@^@^@ was compiled
with
gfortran!^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^
@^@D^@^@^@ÿ^A^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^
@^@^@^@^@^@^A^@^@^@^@GCC: (Ubuntu/Linaro
4.6.4-1ubuntu1~12.04)
4.6.4^@^@^@^@^@^@^@^@^T^@^@^@^@^@^@^@^AzR^@^Ax
^P^A^[^L^G^H<90>^A^@^@^\^@^@^@^\^@^@^@^@^@^@^@
´^@^@^@^@A^N^P<86>^BC^M^F^B¯^L^G^H^@^@^\^@^@^@
<^@^@^@^@^@^@^@;^@^@^@^@A^N^P<86>^BC^M^Fv^L^G^
H^@^@^@^@.symtab^@.strtab^@.shstrtab^@.rela.te
xt^@.data^@.bss^@.rodata^@.comment^@.note.
```

Inspecting Object Files

readelf -a source.o

```
Symbol table '.symtab' contains 18 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.F90
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000   180 FUNC    LOCAL  DEFAULT    1 MAIN__
     7: 0000000000000060    32 OBJECT  LOCAL  DEFAULT    5 options.1.1538
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _gfortran_st_write
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _gfortran_transfer_charac
    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _gfortran_transfer_intege
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _gfortran_st_write_done
    15: 00000000000000b4    59 FUNC    GLOBAL DEFAULT    1 main
    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _gfortran_set_args
    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _gfortran_set_options
```

nm source.o

```
0000000000000000 t MAIN__
                 U _gfortran_set_args
                 U _gfortran_set_options
                 U _gfortran_st_write
                 U _gfortran_st_write_done
                 U _gfortran_transfer_character_write
                 U _gfortran_transfer_integer_write
00000000000000b4 T main
0000000000000060 r options.1.1538
```

Wait… Why do I need to know what’s in my object files?

97% of the time you don’t need to know. However, this can be useful in resolving link errors and multi-language programs

Name Mangling (Fortran)

Fortran

- Binary symbol name is different from high level programming language name
- Variants:
  - Lower, Lower_, Lower__
  - Upper, Upper_, Upper__
- Used to be critical for calling Fortran from C.
  - Now the Fortran standard provides features that give programmer more control over name mangling.

```
SUBROUTINE Sub1(n) BIND(C,NAME="Sub1")
...
ENDSUBROUTINE
```

```
0000000000000000 T Sub1
```

Source File

```
SUBROUTINE Sub1(n)
...
ENDSUBROUTINE
```

Compile

Object File

```
0000000000000000 T sub1_
```

This example is “Lower_”

Name Mangling (C++)

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b)
{
    return a < b ? b:a;
}

int main ()
{
    int i = 39; int j = 20;
    cout << "Max(i, j): " << Max(i, j) << endl;

    double f1 = 13.5; double f2 = 20.7;
    cout << "Max(f1, f2): " << Max(f1, f2) << endl;

    string s1 = "Hello"; string s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;
    return 0;
}
```

- Shows up with templating
  - Have to produce different binary code for each templated type
  - Necessary for linking

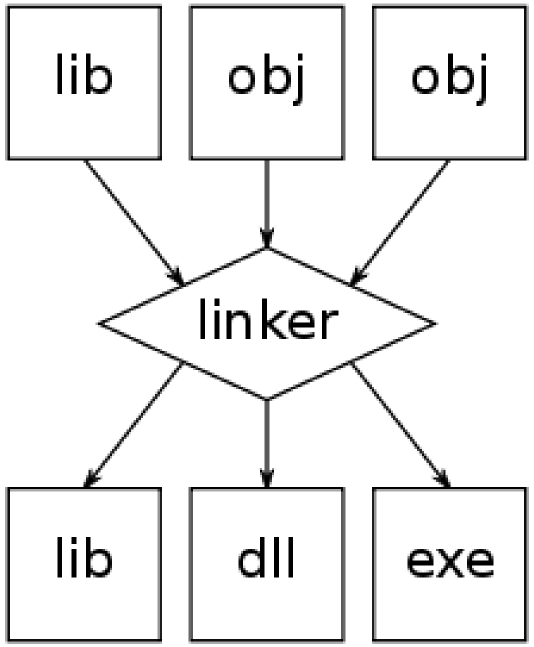
```
00000000000001c6 t _Z41__static_initialization_and_destruction_0ii
                 U _ZNKSs7compareERKSs
                 U _ZNSaIcEC1Ev
                 U _ZNSaIcED1Ev
                 U _ZNSolsEPFRSoS_E
                 U _ZNSolsEd
                 U _ZNSolsEi
                 U _ZNSsC1EPKcRKSaIcE
                 U _ZNSt8ios_base4InitC1Ev
                 U _ZNSt8ios_base4InitD1Ev
                 U _ZSt4cout
                 U _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
0000000000000000 b _ZStL8__ioinit
                 U _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
0000000000000000 W _ZStltIcSt11char_traitsIcESaIcEEbRKSbIT_T0_T1_ES8_
```

Example from: https://www.tutorialspoint.com/cplusplus/cpp_templates.htm

Linking

What is Linking?

- Linking is the process of combining the various objects and libraries output from compilation into a single executable (or library or object).
  - May also include binaries (e.g. libraries) already installed on the system
- Sometimes performed by external program called by compiler (e.g. `ld`)
- Sometimes part of compiler (depends on the vendor)
- Key steps in linking are
  - *Resolving external symbols* that the linker uses to figure out how to piece together the executable
  - *Relocating load addresses* of various program parts (e.g. function addresses and variable addresses) to reflect the assigned addresses in the whole program.
- Linking can produce targets thatare ***statically*** linked or ***dynamically*** linked



Dynamic Linking vs. Static Linking

Static Linking

- Probably what you think of when you think "linking"
- Copy all binary code from all libraries and objects then package into a single executable image
  - Usually results in larger executable file sizes
- A little more portable since all the binary code is packaged together
- Requires all libraries that are linked to be static libraries (e.g. `lib<name>.a`)
- Sometimes a requirement on large clusters
  - Compute nodes and login nodes are different

Dynamic Linking

- Symbol resolution is delayed until executable is run
  - Executable code has undefined symbols
  - Requires all libraries that are linked to be dynamic libraries (e.g. `lib<name>.so`)
- Some advantages
  - For system libraries used by every program, no need to copy into every executable (e.g. `libc`)
  - If there is a bug in a library, and a new version of the library that fixes the bug is installed, all programs benefit.
    - Statically linked executables need to be re-linked
- Some disadvantages
  - Libraries that are updated that break backwards compatibility, might break your executable.
  - Need to have the correct environment.
  - Not necessarily portable, OS and environment need to consistent.

What link errors look like

Static Link Error

```
PROGRAM hello_main

WRITE(*,*) "Hello World!"
CALL some_undefined_routine()

ENDPROGRAM
```

```
$ gfortran -c hello.F90
$ gfortran hello.o -o hello.exe
hello.o: In function `MAIN__':
hello.F90:(.text+0x71): undefined reference to
`some_undefined_routine_'
collect2: ld returned 1 exit status
```

The command given to the linker did not include the library or object (or the correct path to the library or object) that defines the named symbol.

Dynamic Link Error

```
$ ./some_mpi_program.exe
./mpi_program.exe: error while loading shared
libraries: libmpi.so: cannot open shared object file:
No such file or directory
```

When you attempted to run the executable, The OS could not find the library using the information in your current environment

How to trouble shoot link errors

Static Link Error

- Most likely you are missing the correct entries on the following options passed to the linker:
  - `-l<library_name_with_symbol>`
  - `-L<path_to_library>`
- Could also be a typo in your source code
- Generally easy to resolve
  - If you know where the missing library is located.
- Can be difficult if you have no idea why the symbol is trying to be linked (where is it used, where is it defined
  - More likely to happen when you are linking third party libraries

Dynamic Link Error

- Most likely your environment is not the same as when you compiled
  - Check your environment
  - Environment variable is `LD_LIBRARY_PATH`
- Useful command: `ldd`
  - Shows you *exactly* what libraries are dynamically linked to your executable

```
$ ldd ./some_mpi_program.exe
        linux-vdso.so.1 =>  (0x00007ffcf2be8000)
        libmpi.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f4d12878000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f4d12c38000)
```

Dynamic Loading: Linking in code at run time

- Start your executable then load a library into memory.
  - Use case is “plugins”. An example might be linking proprietary correlations for material properties.
  - Can be done interactively. User could specify library name and function name as an input.
  - Challenging to list “available symbols” in library, although this can be done. But basically need to know what routine you want to call
- In Linux requires “dl” library.

```
#include <dlfcn.h>

void* sdl_library = dlopen("libSDL.so", RTLD_LAZY);
if (sdl_library == NULL) {
   // report error ...
} else {
   void* initializer = dlsym(sdl_library,"SDL_Init"); //extract library contents
   if (initializer == NULL) {
      // report error ...
   } else {
      // cast initializer to its proper type and use
      typedef void (*sdl_init_function_type)(void);
      sdl_init_function_type init_func = (sdl_init_function_type) initializer;
   }
}
```

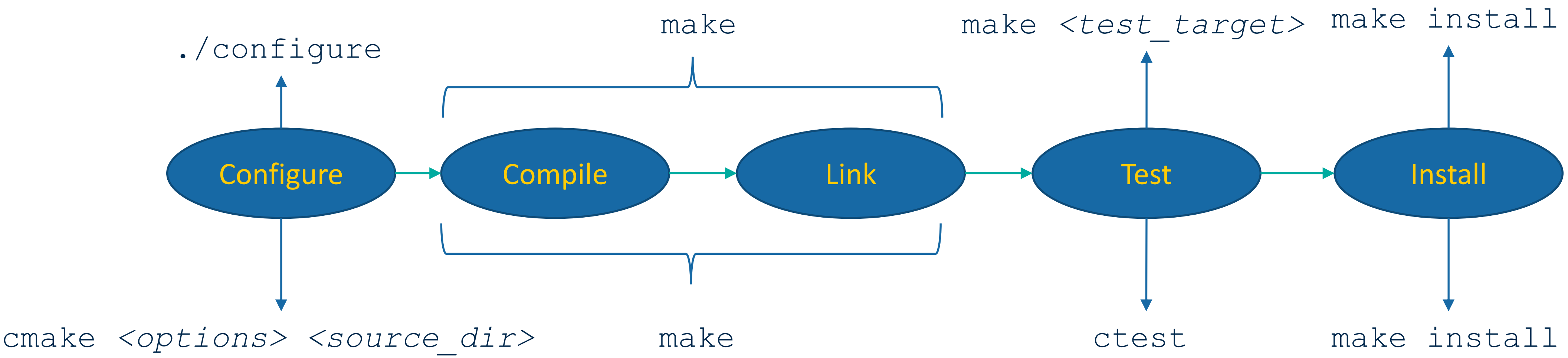
https://en.wikipedia.org/wiki/Dynamic_loading

Multi-language Programs

- **The key is linking!**
  - Linker does not care what high-level language produced your object code. It could have been generated from Fortran or C or C++.
    - Linker just has to resolve symbols in object code.
  - Well one subtlety, you must have a compatible application binary interface (ABI)
    - Usually not an issue unless you are compiling on one machine and linking on another.
- If a programming language or environment (e.g. Python) supports linking of C interfaces than you can link any code that provides a C interface
  - Most languages support C interfaces (because they were probably implemented in C or the compiler was)
  - Therefore, *C is the de-factor language of interoperability*.
- By “C interface” I mean a binary symbol that is producible from the C high-level language and a C compiler.

Summary: Using the Toolchain

Autotools



CMake