Introduction to Compiling and Debugging

Drew Gustafson
(dgustaf@msi.umn.edu)
Evan Bollig
(evan@msi.umn.edu)

Hands-on

`http://z.umn.edu/compile`

Tutorial Summary

- Building Software
  - Compiling and Linking
  - Automated Build Systems
- Debugging Code
- Profiling
- Hands-On

Compiling and Linking

...

Compiler Phases, Optimization Flags, Link Libraries, etc.

What is a Compiler?

A compiler is a program to turn human readable source code into a machine code "program" for execution.

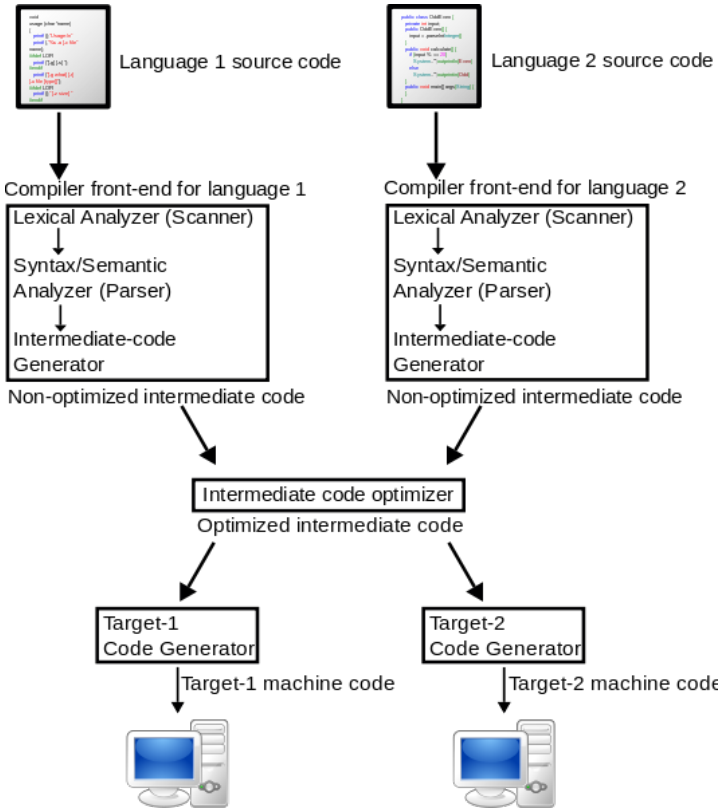


Image source: http://en.wikipedia.org/wiki/File:Compiler.svg

- Not all programming languages use compilers; the fastest executing ones do.
- In the process of compiling, most compilers partially "optimize" code for faster, more efficient execution.

The Compilers Available @MSI

Intel Compilers

- Loaded with: module load intel
- C compiler: icc
- C++ compiler: icpc
- Fortran compiler: ifort
- Very good optimization and integration with Intel libraries like MKL
- Can see available versions with:
- module avail intel

The Compilers Available @MSI

GNU Compilers

- Loaded with: module load gcc
- C compiler: gcc
- C++ compiler: g++
- Fortran compiler: gfortran
- Free and open source
- Good optimization and integration with open source libraries
- Can see available versions with:
- module avail gcc

The Compilers Available @MSI

PGI Compilers

- Loaded with: module load pgi
- C compiler: pgcc
- C++ compiler: pgcpp
- Fortran compiler: pgfortran
- Supports some different code syntax
- Can see available versions with:
- module avail pgi

Compiling Commands

- The usual format for compiling commands is:
  - **$> compilername flags sourcefiles**
    - **compilername** should be the name of the compiler being used.
    - **flags** are optional commands (called flags) that change compiling options. Most flags begin with the minus sign (-).
    - **sourcefiles** are the names of the source code files that are being compiled.
- Example command for compiling C++ code with the Intel C++ compiler:

  $> icpc -o my.exe mysourcecode.cpp

By default most compilers name the compiled program **a.out**

Useful Compiler Options (Flags)

- To specify a name for the compiled program use the -o flag followed by the desired program name.

- Intel Compiler Examples:

$> icpc -o myprogram mysourcecode.cpp or
$> icc -o myprogram mysourcecode.c or
$> ifort -o myprogram mysourcecode.f

- GNU Compiler Examples:

$> g++ -o myprogram mysourcecode.cpp or
$> gcc -o myprogram mysourcecode.c or
$> gfortran -o myprogram mysourcecode.f

- For multiple source files the command is like:

$> g++ -o myprogram mysourcecode1.cpp mysourcecode2.cpp

Optimization Flags

- The compiler can attempt to perform some automatic optimizations of the source code, and this can often tremendously improve program speed.

- The flags controlling optimization begin with **-O** (oh, not zero), and most compilers support three options: **-O1**, **-O2**, **-O3**
- The **-O3** flag is the most aggressive optimization, while **-O1** is least aggressive.

- The automatic optimizations can remove unnecessary portions of code, and reorganize code so that it performs more efficiently.

- Using optimization flags can sometimes slightly alter the output of a program. Usually any difference will occur in the smallest digits of numbers being calculated, and will only be significant if the program is very sensitive to such values.

Compiling Using Object Files

- It is sometimes useful to compile in steps, first compiling to “object” (.o) files, and then “linking” the object files.

- To compile to object files use the -c flag:

$> icpc -c mysourcecode1.cpp mysourcecode2.cpp

  - This will produce files named mysourcecode1.o and mysourcecode2.o

- To perform the linking use the command:

$> icpc -o myprogram mysourcecode1.o mysourcecode2.o

  - This will “link” the object files and create an executable named myprogram.

Compiling Using Object Files

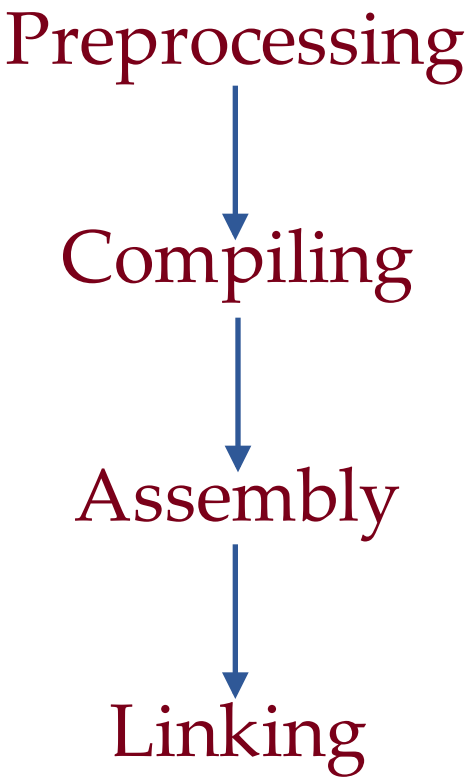
Compiling using object files has some advantages and disadvantages:

- It reduces the time to recompile after making small changes to a large program. In such a case only a few of many object files will need to be recompiled. It also reduces compile time in many files have shared dependencies.

- It makes it easier to work collaboratively on code, with each coder compiling their own object files which are part of a larger program.
- It allows the linking of object files that were created via different programming languages (care must be taken with this).

- It can prevent some compiler optimizations because it makes it harder for the compiler to determine how all the pieces fit together. This effect is usually minor.

Compiling Occurs in Steps



Parses and alters code

Translates into Assembly language

Translates into machine code

Links machine code pieces and libraries into final machine code program

It is possible to stop at intermediate stages.

Preprocessor Commands

Preprocessor statements within program code can be used to include other code and insert constants and code snippets. Most compilers use # to indicate a preprocessor directive.

Include statements insert code:

#include <cmath>

- This will cause the preprocessor to insert the contents of the cmath file at this place in the code.

Define statements insert constants or code snippets:

#define Pi 3.141592653

- This will cause the preprocessor to replace every instance of Pi with the numerical representation.

Linking with Libraries

Libraries are files containing frequently used machine code. They can be used in a program, and are incorporated during the linking step.

Libraries often contain commonly used functions. An example is the FFTW Fourier Transform library.

The libraries to link with are specified in the compile command:

$> g++ -o myprogram mysourcecode.cpp –lfftw3 `or`

$> g++ -o myprogram mysourcecode.o –lfftw3

- The library linking flags begin with –l
- The library files themselves usually have names beginning with lib, for example: **libfftw3.so**

Linking with Libraries

Library files come in two types: dynamic and static

- Dynamic library files have names ending in **.so**
- Static library files have names ending in **.a**

Dynamic libraries remain distinct from programs, and are loaded by a program during execution when they are needed. Dynamic libraries can be upgraded without recompiling the programs using them (within limits).

Static libraries become part of the program using them. When a static library is used while compiling the program generated will always load that version of the library.

Where the Compiler Searches for Files

The compiler needs to locate:

- Source code files
- Files included via preprocessor statements
- Library files

The compiler will search for source code files in the place the compiling command is executed.

Include files and library files are looked for in a number of locations governed by compiling options.

Where the Compiler Searches for Files

Include files are files containing code (often "headers") that will be inserted into the source code by the preprocessor.

Places specified in the compiling command using the -I flag, example:

$> gcc -I/soft/fftw/include –o myexe source.c

- Places referenced by environmental variables: CPATH, FPATH, C_INCLUDE_PATH, CPLUS_INCLUDE_PATH, INCLUDE
- The variables are mostly C, C++, Fortran specific
- "Default" locations like: /usr/include, etc

Where the Compiler Searches for Files

Library files are files containing machine code that the program will link to during the linking step of compiling.

Places specified in the compiling command using the -L flag, example:

$> gcc -L/soft/fftw/lib –o myexe source.c

- Places referenced by environmental variables: LD_LIBRARY_PATH, and sometimes LIBRARY_PATH
- "Default" locations like: /usr/lib, /usr/lib64, etc.

Where the Compiler Searches for Files

- It is possible to alter where the compiler searches for files by altering environmental variables:

  $> export FPATH=/soft/fftw/include:$FPATH

  - This would make the fortran compiler search within /soft/fftw/include for include files.
  - Equivalent for Fortran compile statements:

  $> ifort -I/soft/fftw/include mysourcecode.f

- For C/C++ and generally any other non-FORTRAN language:

  $> export LD_LIBRARY_PATH=/soft/fftw/lib:$LD_LIBRARY_PATH

  - Equivalent at compile time:

  $> icc -L/soft/fftw/lib mysourcecode.c

Additional Tips

When a compiler module is loaded it makes changes to the environment, mostly by altering environmental variables. To see what a module does use the command:

$> module show modulename

- By examining what changes the module makes you can determine where the library and header files are located. For compiling some complex programs you may need to specify these locations using the -L and -I flags.
- Sometimes compiling is sensitive to the order in which the library files are linked. If linking with libraries seems to fail you may need re-order the linking flags in the compile command.

Building Large Projects

Makefiles, CMake, Autoconf, etc.

Build Systems

- Most codes are complex with interdependent source files.
- Individual files need to be compiled into object files (.o) then linked into a binary (use the -c option)

```
$> gcc -c foo.c -o foo.o
$> gcc -c bar.c -o bar.o
$> gcc foo.o bar.o -o foobar.exe
```

  - NOTE: Linking is required to satisfy **undefined external references** (common compile error)
- At scale (10+ files), this becomes cumbersome
  - We need automation!

Make

What is Make?

- GNU Make "*is a tool which controls the generation of executables and other non-source files of a program from the program's source files.*" [1]
- Automatically determines which parts to (re)build
- Most popular build utility
  - Available by default on most Unix/Linux machines
  - Cross platform options (Windows, OSX)

Using a Makefile

- Executing Make: **`$> make`**
- Make looks for a file named **makefile** or **Makefile** in the current directory
  - Specify a different file: **`$> make -f other.mk`**
- Specify a target: **`$> make test1.exe`**

What is a Makefile?

- Input file specifically for the GNU Make utility
  - Similar to a shell script
- Specifies targets, rules and dependencies for each part of build
  - Targets - callable scriptlets that have dependencies and rules
  - Rules - build tasks/actions (command lines)
  - Dependencies - control target call order

Why use Make?

- Makefile keeps track of which files need to be recompiled
  - Uses "Date Modified" timestamp to know when changes are made to any files
  - If a single dependency changes it is recompiled, plus all dependent sources
- Run one command and let Make worry about each call to compile and link:

```
$> make
```

Makefile Example (1)

- Simple Target:

```
helloworld.exe:
        gcc helloworld.c -o helloworld.exe
```

- **TAB** (not spaces) before gcc is required!
  - May require modification of your .vimrc / editor settings
- Specify which target to build:

```
$> make helloworld.exe
```

- First target in Makefile is executed by default:

```
$> make
```

Makefile Example (2)

- Compile and Link

```
foobar.exe:
    gcc -c foo.c -o foo.o
    gcc -c bar.c -o bar.o
    gcc foo.o bar.o -o foobar.exe
```

- Each target can contain a full set of rules written like shell scripts (Bash-style)
- Tip: use **@echo** to print text without seeing the original command

Adding Dependencies

```
foobar.exe:  foo.o bar.o
    gcc foo.o bar.o -o foobar.exe

foo.o:  foo.c
    gcc -c foo.c -o foo.o

bar.o:  bar.c
    gcc -c bar.c -o bar.o
```

- `foo.c -> foo.o ->`
- `bar.c -> bar.o -> foobar.exe`
- We depend on sources so they are recompiled when edited

Macros/Variables

User-specified- and environment-variables work the same as in shell scripting:

```
CC=gcc
CCFLAGS = -O0
COMPILE = -g
COMPILE += $(CCFLAGS)
foo.o: foo.c
   $(CC) $(COMPILE) -c foo.c -o $(HOME)/foo.o
```

- CC, CCFLAGS and COMPILE are local variables, HOME comes from the Unix environment

Predefined Variables

These variables are predefined by Make, but can be overridden in your Makefile (http://www.gnu.org/software/make/manual/make.html#Implicit-Variables):

- **CC** -- The C compiler (default 'cc' -- not always equiv. to 'gcc')
- **CXX** -- The C++ compiler (default 'g++')
- **FC** -- The Fortran compiler (default 'f77' -- requires override!!)
- **AR** -- The archive or library binder (default 'ar')
- **CFLAGS** -- Options, flags and defs for C compiler
- **CXXFLAGS** -- Options, flags and defs for C++ compiler
- **FFLAGS** -- Options, flags and defs for Fortran compiler
- **LDFLAGS** -- Options, flags and defs for linking stage

Abstracting with Variables

- The variable **$@** is used inside a rule and stands for the name of the target

  gcc -c foo.c -o foo.o <=> gcc -c foo.c -o $@

- Variable **$^** represents target's list of dependencies

  gcc foo.o bar.o -o foobar.exe <=> gcc $^ -o $@

- More complicated:
  - **$<** gets the first prerequisite to the right of the ":" or the first one added by an implicit function
  - **$*** gets the stem of a target pattern match (e.g. if given foo.c, **%.c** matches and sets **$*** to "foo")

Advanced Make

- .SUFFIXES are for old Makefile support:

```
CC=gcc
CFLAGS=-g
.SUFFIXES= .c .o

foobar.exe: foo.o bar.o
	$(CC) $^ -o $@

.c.o:
	$(CC) -c $(CFLAGS) $< -o $@
```

- Suffix rules CANNOT have prerequisites

Libraries and Modules (1)

- Use Make to build a STATIC library (lib<name>.a):

```
libFoobar.a: libFoobar.a(foo.o bar.o)

(%.o): %.c
	$(CC) $(CFLAGS) -c $< -o $*.o
	$(AR) rc $@ $*.o
```

- Express members of archive/library inside ( )'s. Note that **$@** in second target is "libFoobar.a" instead of "(foo.o)"

Libraries and Modules (2)

- Build SHARED Library (lib<name>.so) with Make:

```
libFoobar.so: foo.o bar.o
  $(CC) $(LDFLAGS) -shared $^ -o $@
```

- Uses compiler instead of archive utility (ar)
- What is the difference between .a and .so?

Libraries and Modules (3)

- Build a F90 MODULE (.mod):

```
FC= gfortran
FFLAGS= -g
foobar.exe:  foo.o bar.o
    $(FC)  $(FFLAGS)  $^ -o $@
bar.o:  foo.mod
%.mod %_mod.mod:  %.f90
    @if [ ! -e $@ ]; then \
      rm -f $*.o; \
      $(MAKE)  $*.o; \
    fi
%.o:  %.f90
    $(FC)  $(FFLAGS)  $< -o $@
```

Libraries and Modules (4)

- Notes on previous slide:
  - foo contains a module. A .mod file is generated when foo is compiled. However, we must express dependencies so .mod exists before compiling bar.
  - The target %.mod %_mod.mod: %.f90 tests if the .mod file exists. If not, it forces the .f90 to be recompiled to produce a new .mod and .o.
  - %_mod.mod is included becuase many people define [name]_mod inside [name].f90

Libraries and Modules (5)

Create a Java JAR (very similar to creating static lib):

```
JC = javac
JFLAGS = -g
CLASSES = $(patsubst %.java,%.class,$(wildcard
   *.java))
foobar.jar: $(CLASSES)
    touch $@
    jar uvf $@ $(^:.class=*.class)

%.class: %.java
    $(JC) $(JFLAGS) $^
```

Libraries and Modules (6)

- Notes on previous slide:
  - We touch the jar to ensure it exists when we update/append files to it (jar uvf)
  - $(^:.class=*.class) substitutes *.class for .class in the dependency list
    - Required since multiple classes inside [name].java produces [name].class, [name]$[class2].class, ..., [name]$[classN].class

.PHONY Targets

- .PHONY defines targets with no meaning as files, only as commands to execute

```
.PHONY: clean
clean:
	rm *.o *.exe
```

- Two reasons to use: avoid name conflicts and trim pattern matching tree

Build Multiple Directories

Use .PHONY, inheritance and dependency to build subdirectories:

```
SUBDIRS = foo bar baz
.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
    $(MAKE) -C $@

foo: baz
```

Other Tips and Tricks

- Specify Variables at Compile Time:
- **make "CC=ifort"**
- Build in Parallel: **make -j <# of Threads>**
  - Dependencies must be clearly defined or build will fail on race condition
- Echo, Don't Execute (Dry Run): **make -n**
  - Lets you verify build order, compiler flags and search paths

More Information...

- [1] The Official GNU Make Manual: http://www.gnu.org/software/make/manual/make.html
- `man make`

CMake

Why CMake?

- CMake is a cross platform Makefile generator
- We tell it our end goal and CMake fills in the rest
  - "the rest" includes searching for dependencies (libs, headers, etc), building dependency tree and setting the right compiler flags
  - It does NOT build your project. It only creates a Makefile

Do I need a Makefile?

- **No.** But you do need a CMakeLists.txt file, and the cmake command (module load cmake)
  - If you see this in 3rd party software, the build process is standard

- Build process:

```
cmake .
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc – works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info – done
 […]
```

CMake Succeeds…Now what?

- A Makefile was generated. Run make
- and/or make install and treat it like the build system we know.
- Where does CMake install?
  - CMAKE_INSTALL_PREFIX

```
cmake -DCMAKE_INSTALL_PREFIX=<install_path> .
make
make install
```

More Information

- https://cmake.org/cmake-tutorial/
  - A formal CMake lecture, including how to write your own CMakeLists.txt

GNU Configure

Configure Script

- Many distributions are built using the standard:

```
./configure
make
make install
```

- Same concept as CMake: let it generate your Makefile

- Specify the Install Path:

```
./configure --prefix=<install_path>
```

More Information

- http://en.wikipedia.org/wiki/Configure_script
- http://www.airs.com/ian/configure/

Debugging Code

GDB, Valgrind, etc.

Debugging

segfault

- A Segmentation Fault (segfault) occurs when a routine attempts to access invalid memory addresses
  - Memory outside scope of routine
  - Unaligned memory addresses
- Result: execution is terminated by the kernel
  - Effects may be delayed.

Buffer Overflow

- A Buffer Overlow occurs when a routine write past the end of a buffer/array into adjacent memory.
- Not guaranteed to issue segfault or other warning

Simple Debugging

- Use print statements to find where your code breaks
  - Binary search to find range of lines
- Debugger: gdb
  - Requires –g flag on compile

```
gdb ./a.out
(gdb) run
(gdb) p c[10]
```

GDB Commands

- where – print line where segfault occurred
- break <line> – set a breakpoint (pause in code) at line
- list <line> -- print lines of code in the neighborhood of line
- p <var> (or) print <var> -- print the current value of variable
  - p array[10]@100 – print 100 elements of array starting with element 10
- cont (or) continue – continue execution to the next breakpoint
- step – execute the next line of code with a one-time breakpoint

Memory Checking

Valgrind

Valgrind is a GNU debugging tool that specializes in memory checking. Valgrind is good at finding memory leaks, or errors involving pointers. To use Valgrind to debug first compile your code with a GNU compiler using the -g flag:

```
$> g++ -g -o myprogram mysourcecode.cpp
```

Then execute the program within Valgrind using a Valgrind tool. Memcheck is the most commonly used Valgrind tool, and it is good at finding pointer errors:

```
$> valgrind --tool=memcheck ./myprogram
```

Valgrind can report the place in the source code where pointer or memory errors occur. Some often reported pointer errors are “Invalid read” and “Invalid write”.

Profiling

Why Profile?

- Most codes follow the 90-10 standard
  - 90% time spent in 10% of the code
- **Don't waste 90% of your time optimizing code that accounts for 10% of the execution**
  - Profilers give you an idea of where the hotspots are based on a single execution

GProf

- Compile with **-pg**
- Execute your binary once
  - This writes a gmon.out file
- View the profile with

```
$> gprof <exec_name>
```

Hands-on

• • •

```
git clone
https://github.umn.edu/msi/tutorial_intro_compili
ng_debugging_fall2015.git
```