Introduction to Cluster Computing:

Linux, shell scripting, queuing systems, cluster architecture

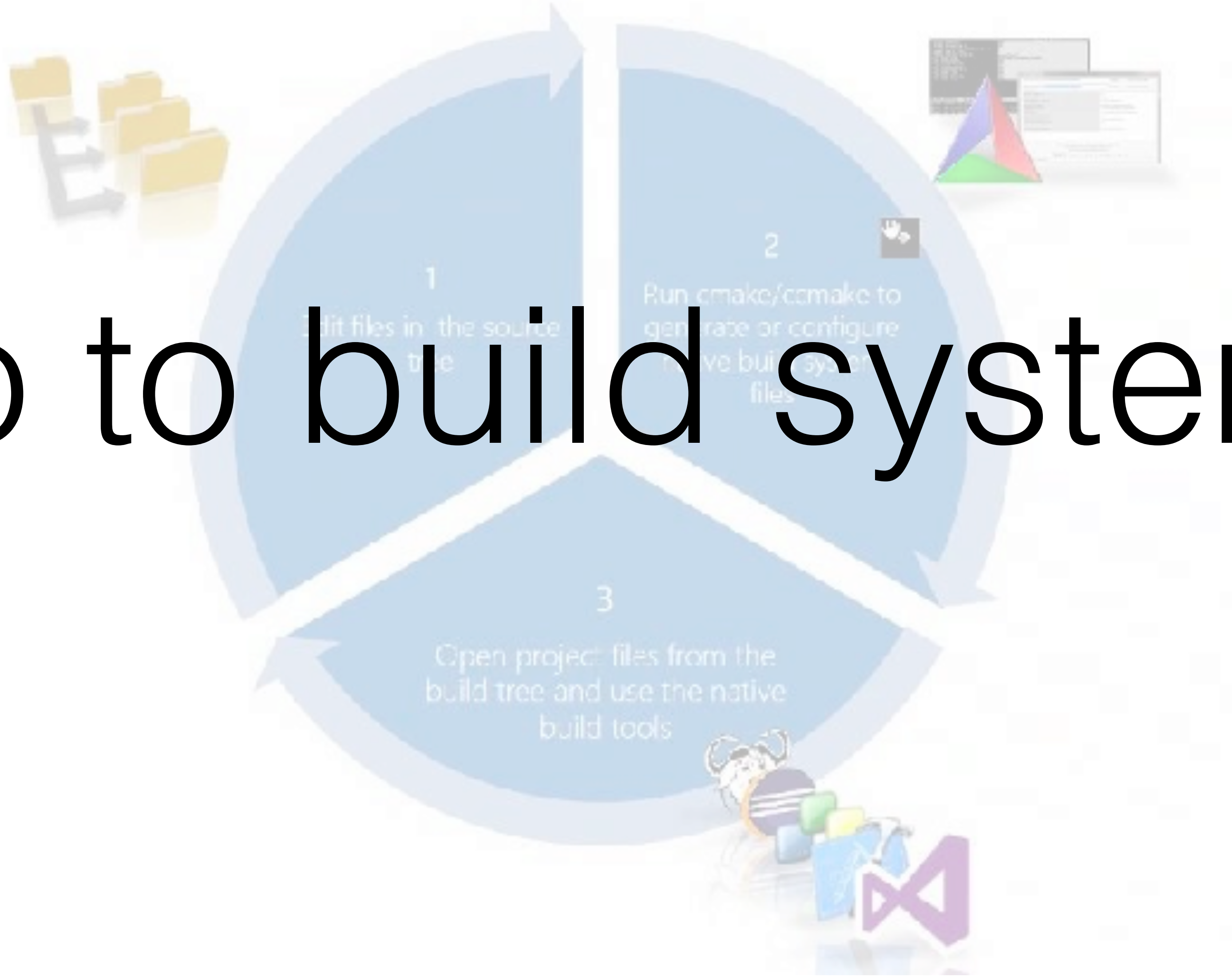
Instructor: Dr. Peggy Lindner (plindner@uh.edu)

Lecture 5 (git, make and the Linux environment)

UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Intro to git

Intro to build systems


1
Edit files in the source tree
2
Run cmake/ccmake to generate or configure native build system files
3
Open project files from the build tree and use the native build tools

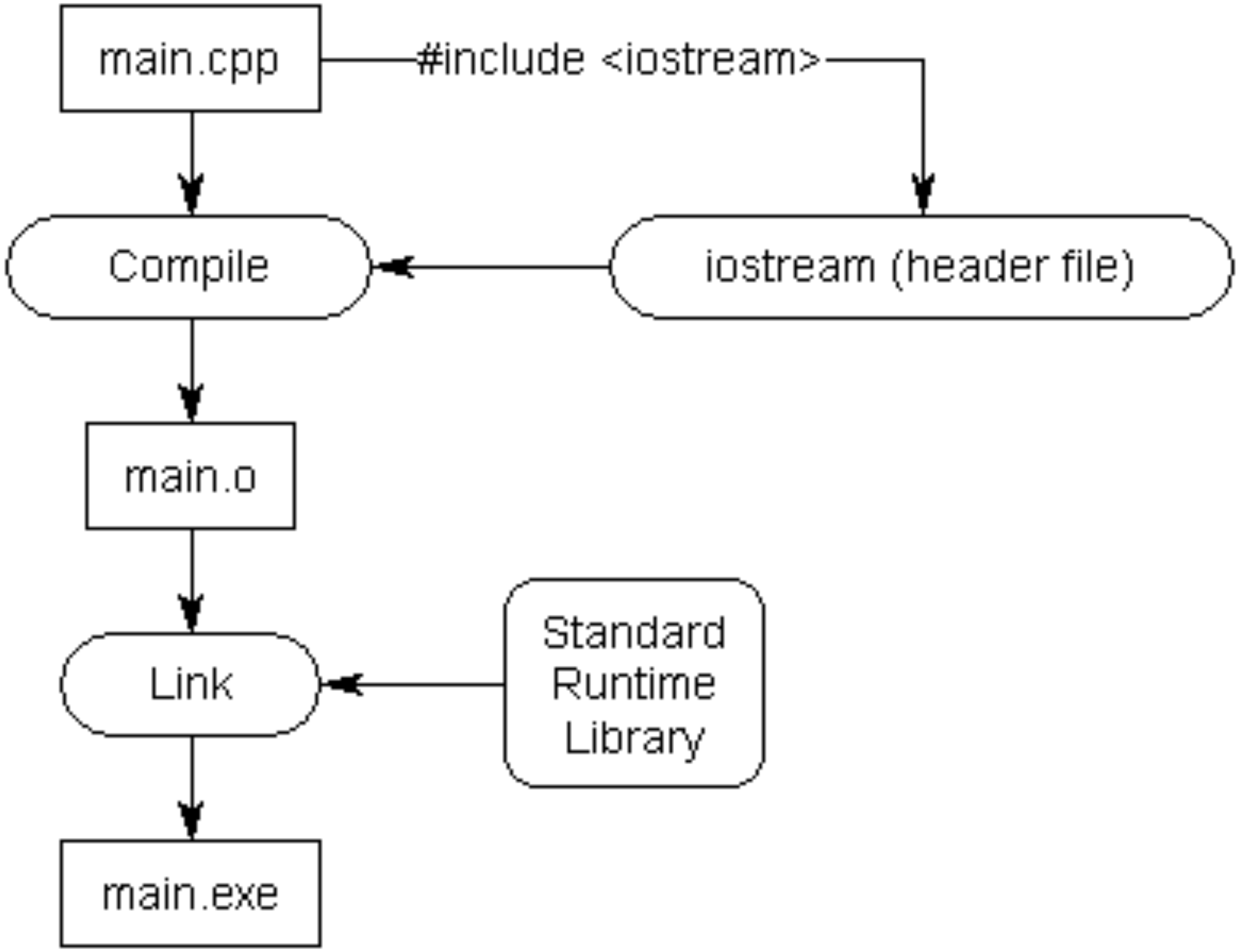
Recap Compilation

- A compiler turns human-readable source code into machine-readable object code that can actually run.

- Compiler of choice for most Linux systems is GCC (GNU Compiler Collection)
  – gcc, g++, gfortran

- Other available compilers on HPC Systems include
  – Intel Compiler Suite (icc, icpc, ifort)
  – PGI (Portland Group) (pgcc, pgCC, pgf77, pgf90, pgfortran)

Compilation Flow



Compilation Example

- Assuming our header files are in <u>same folder</u>

```
$ cd intro2linux_make/reciprocal

#compile the objects
$ g++ -c main.cpp
$ g++ -c reciprocal.cpp


#link the objects
$ g++ -o reciprocal reciprocal.o main.o

#run the application
./reciprocal 7
The reciprocal of 7 is 0.142857
```

Compilation Example

- Assuming our header files are in the include folder

```
$ cd intro2linux_make/reciprocal

#compile the objects
g++ -c -I ./include main.cpp
g++ -c -I ./include reciprocal.cpp

#link the objects
$ g++ -o reciprocal reciprocal.o main.o

#run the application
./reciprocal 7
The reciprocal of 7 is 0.142857
```

Compilation Example

- Assuming our header files are in the include folder

- “g++” links reciprocal to the standard C++ library containing *cout*.

- To see linked libraries use the ldd command

  ```
  ldd ./reciprocal
  ```

- To link to additional libraries
  - we use the “-l” + ’library_name’ option

    ```
    g++ -o reciprocal reciprocal.o main.o -lm
    ```

  - -lm => link to gnu C math library (libm.so)

Why make?

- Building executables from 2 source files is ok while using g++ from command line.
- However, its impractical to use gcc from command line building for large projects (dozens to thousands of source codes files)
- Linux developers automated the building larger source code projects using “GNU make” or make in UNIX

What is *make*?

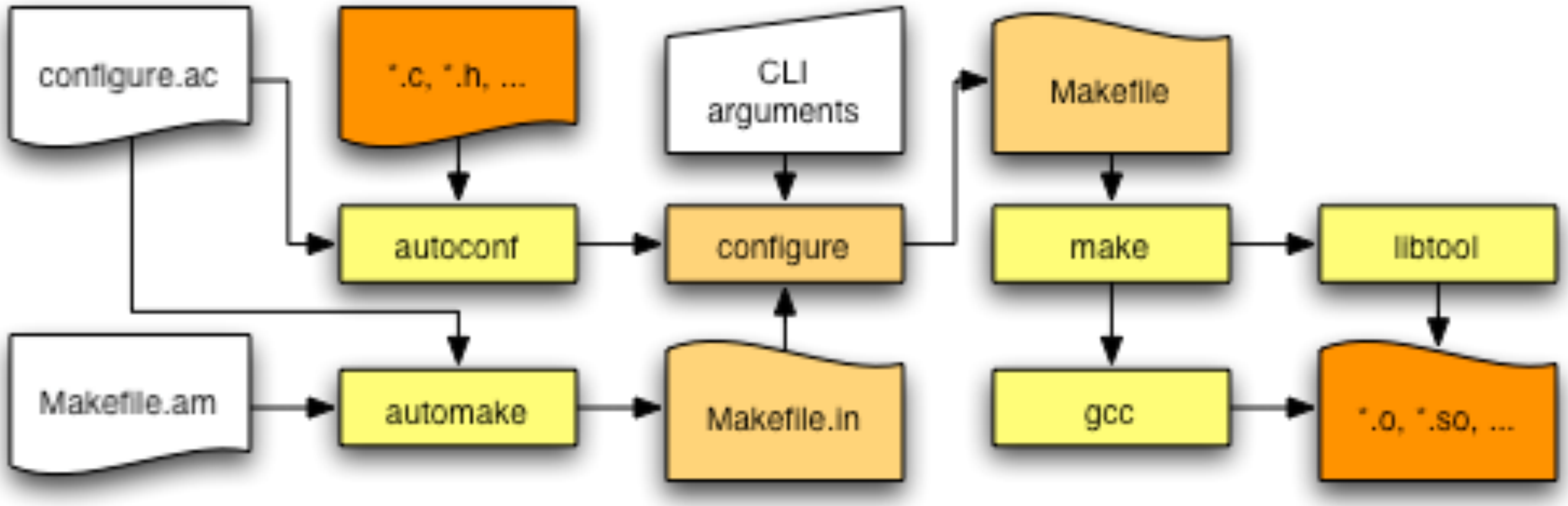
- Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.

- Information for building the project is conveyed to make using a `Makefile`

- A `Makefile` is a text file containing specification of dependencies between files and how to resolve those dependencies such that an overall goal, known as a target, can be reached. ‘Makefile’s are processed by the make utility.

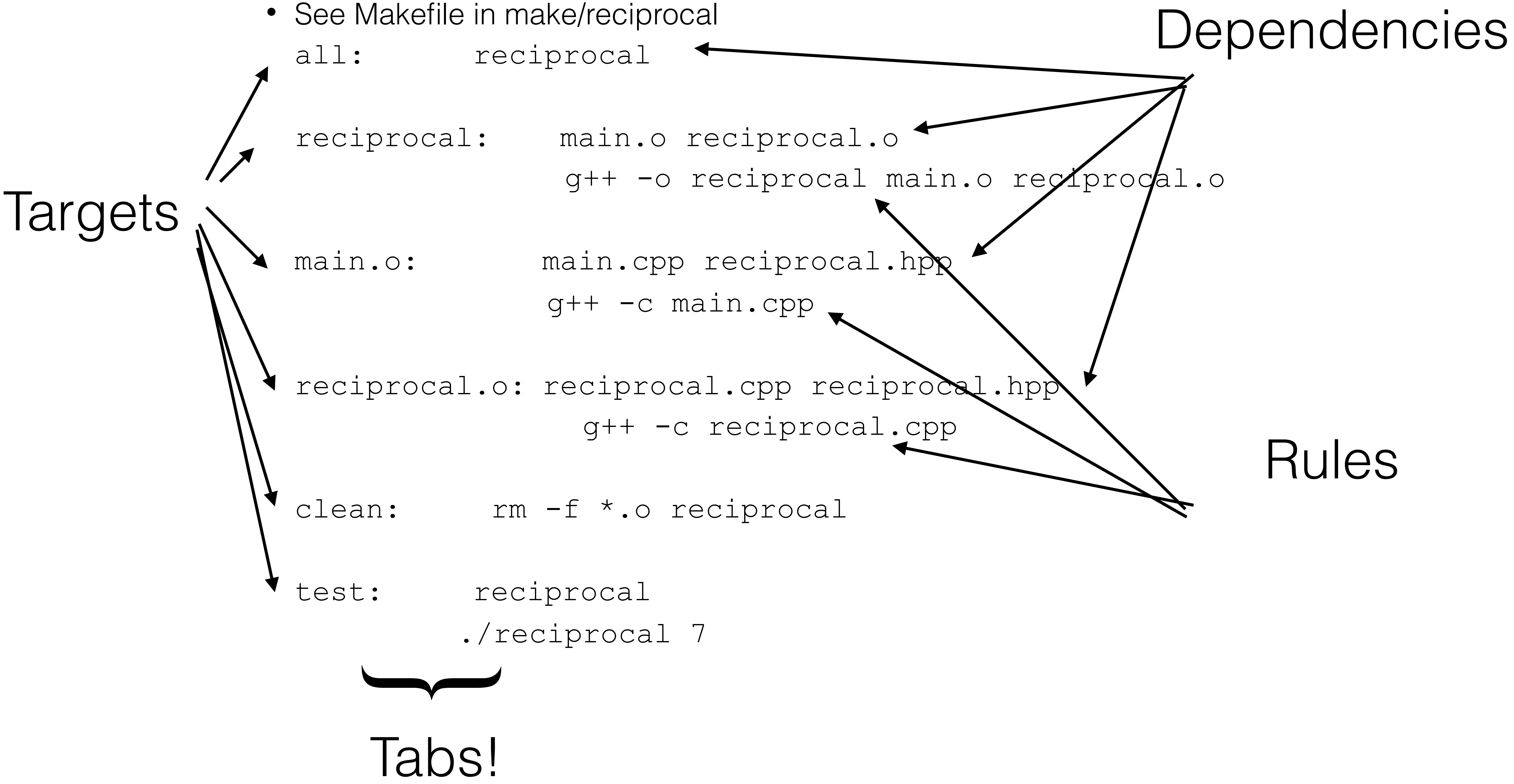
What is *make*?

- Build System



Example

- See Makefile in make/reciprocal



```
all:       reciprocal

reciprocal:    main.o reciprocal.o
               g++ -o reciprocal main.o reciprocal.o

main.o:        main.cpp reciprocal.hpp
               g++ -c main.cpp

reciprocal.o: reciprocal.cpp reciprocal.hpp
                g++ -c reciprocal.cpp

clean:     rm -f *.o reciprocal

test:      reciprocal
          ./reciprocal 7
```

Targets

Dependencies

Rules

Tabs!

Running the Example

```
cacds25@whale:~/intro2linux_make/make/
reciprocal> make

g++ -c main.cpp

g++ -c reciprocal.cpp

g++ -o reciprocal main.o reciprocal.o

cacds25@whale:~/intro2linux_make/make/
reciprocal>
```

Build Command

Running the Example

```
cacds25@whale:~/intro2linux_make/make/reciprocal> make clean

rm -f *.o reciprocal

cacds25@whale:~/intro2linux_make/make/reciprocal> make test

g++ -c main.cpp

g++ -c reciprocal.cpp

g++ -o reciprocal main.o reciprocal.o

./reciprocal 7

The reciprocal of 7 is 0.0588235

cacds25@whale:~/intro2linux_make/make/reciprocal>
```

Make and Macros

- We can define options within a Makefile, e.g. here we define compiler (CXX) and compiler optimization variables (CXXFLAGS), see `Makefile_tune`

```
CXXFLAGS=-O3

CXX=g++

all:        reciprocal

reciprocal: main.o reciprocal.o

             $(CXX) $(CXXFLAGS) -o reciprocal main.o reciprocal.o

main.o:     main.cpp reciprocal.hpp

             $(CXX) $(CXXFLAGS) -c main.cpp

reciprocal.o:   reciprocal.cpp reciprocal.hpp

             $(CXX) $(CXXFLAGS) -c reciprocal.cpp

clean:

       rm -f *.o reciprocal

test:       reciprocal

       ./reciprocal 7
```

Further notes on *make*

- Variables make Makefiles simpler

- It is standard practice for every makefile to have a variable named objects, OBJECTS, objs, OBJS, obj, or OBJ which is a list of all object file names. We would define such a variable objects with a line like this in the makefile:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

Running the Example

```
cacds25@whale:~/intro2linux_make/make/reciprocal> make -f Makefile_tune

make: Nothing to be done for `all'.

cacds25@whale:~/intro2linux_make/make/reciprocal> make clean

rm -f *.o reciprocal

cacds25@whale:~/intro2linux_make/make/reciprocal> make -f Makefile_tune

g++ -O3 -c main.cpp

g++ -O3 -c reciprocal.cpp

g++ -O3 -o reciprocal main.o reciprocal.o

cacds25@whale:~/intro2linux_make/make/reciprocal> make -f Makefile_tune clean

rm -f *.o reciprocal

cacds25@whale:~/intro2linux_make/make/reciprocal> make -f Makefile_tune test

g++ -O3 -c main.cpp

g++ -O3 -c reciprocal.cpp

g++ -O3 -o reciprocal main.o reciprocal.o

./reciprocal 7

The reciprocal of 7 is 0.0588235

cacds25@whale:~/intro2linux_make/make/reciprocal>
```

Reference to new Makefile

Unix command diff

- The UNIX diff command compares the contents of two text files and outputs a list of differences `diff [options] file1 file2`.

```
cacds25@whale:~/intro2linux_make/make/reciprocal> diff Makefile_tune2
Makefile_tune3
3c3
< CC=c++
---
> 
7,8c7,13
< main.o:    main.cpp reciprocal.hpp
<     $(CXX) $(CXXFLAGS)  -c $<
---
> .cpp.o:    $(CXX) $(CXXFLAGS) -c $< -o $@
>
> .c.o:   $(CC) $(CFLAGS) -c $< -o $@
>
> .f90.o:         $(F90FLAGS) $(F90FLAGS) -c $< -o $@
>
> .f77.o:         $(FFLAGS) $(FFLAGS) -c $< -o $@
10,11d14
< reciprocal.o:reciprocal.cpp  reciprocal.hpp
<     $(CXX) $(CXXFLAGS) -c $<
```

File 1

File 2

Make and automatic variables

- Automatic variables have values computed afresh for each rule that is executed, based on the target and prerequisites of the rule. e.g. here we call to first dependency with ‘$<’ (for the source file name), *all dependencies* with ‘$?’ and target (the object file name) with ‘$@’, see `Makefile_tune2`

```
CXXFLAGS=-O2

CXX=c++

CC=c++

all:           reciprocal

main.o:        main.cpp reciprocal.hpp

               $(CXX) $(CXXFLAGS)  -c $<

reciprocal.o: reciprocal.cpp  reciprocal.hpp

                $(CXX) $(CXXFLAGS) -c $<

reciprocal:    main.o reciprocal.o

               $(CXX) $(CXXFLAGS) $? -o $@

clean:

                rm -f *.o reciprocal

test:          reciprocal

               ./$< 7
```

Make and automatic variables

- Automatic variables have values computed afresh for each rule that is executed, based on the target and prerequisites of the rule. e.g. here we call to first dependency with ‘$<’ (for the source file name), *all dependencies* with ‘$?’ and target (the object file name) with ‘$@’, see `Makefile_tune3`

```
CXXFLAGS=-O2
CXX=c++
all:          reciprocal
.cpp.o:       $(CXX) $(CXXFLAGS) -c $< -o $@
.c.o:         $(CC) $(CFLAGS) -c $< -o $@
.f90.o:         $(F90FLAGS) $(F90FLAGS) -c $< -o $@
.f77.o:         $(FFLAGS) $(FFLAGS) -c $< -o $@
reciprocal:   main.o reciprocal.o
        $(CXX) $(CXXFLAGS) $? -o $@
clean:
        rm -f *.o reciprocal

test:         reciprocal
        ./$< 7
```

Developing libraries Example

- see `Makefile in folder make/svd`

```
MKL_LIB_ROOT=/share/apps/intel/mkl/lib/intel64/

MKL_LIB_ROOT=/opt/intel/mkl/lib/intel64/

LIBS= -L${MKL_LIB_ROOT} -Wl,-rpath,${MKL_LIB_ROOT} -lmkl_intel_lp64 -lmkl_core -lmkl_intel_thread -lpthread -lm -ldl

CC=icc

CFLAGS= -openmp

all:          svd

svd:          svd.o

        $(CC) $(CFLAGS) $< -o $@ $(LIBS)

svd.o:        svd.c

        $(CC) $(CFLAGS) -c $<

clean:

        rm -f *.o svd

test:         svd

        ./svd
```

Opuntia only!

Developing MPI Example

- see `Makefile in folder make/mpi`

```
CC=mpicc

all:     gethostname

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

gethostname: gethostname.o

      $(CC) $< -o $@

clean:

      rm -f *.o gethostname

test:    gethostname

      mpirun -np 4 ./$<
```

Installing software with make in your home directory

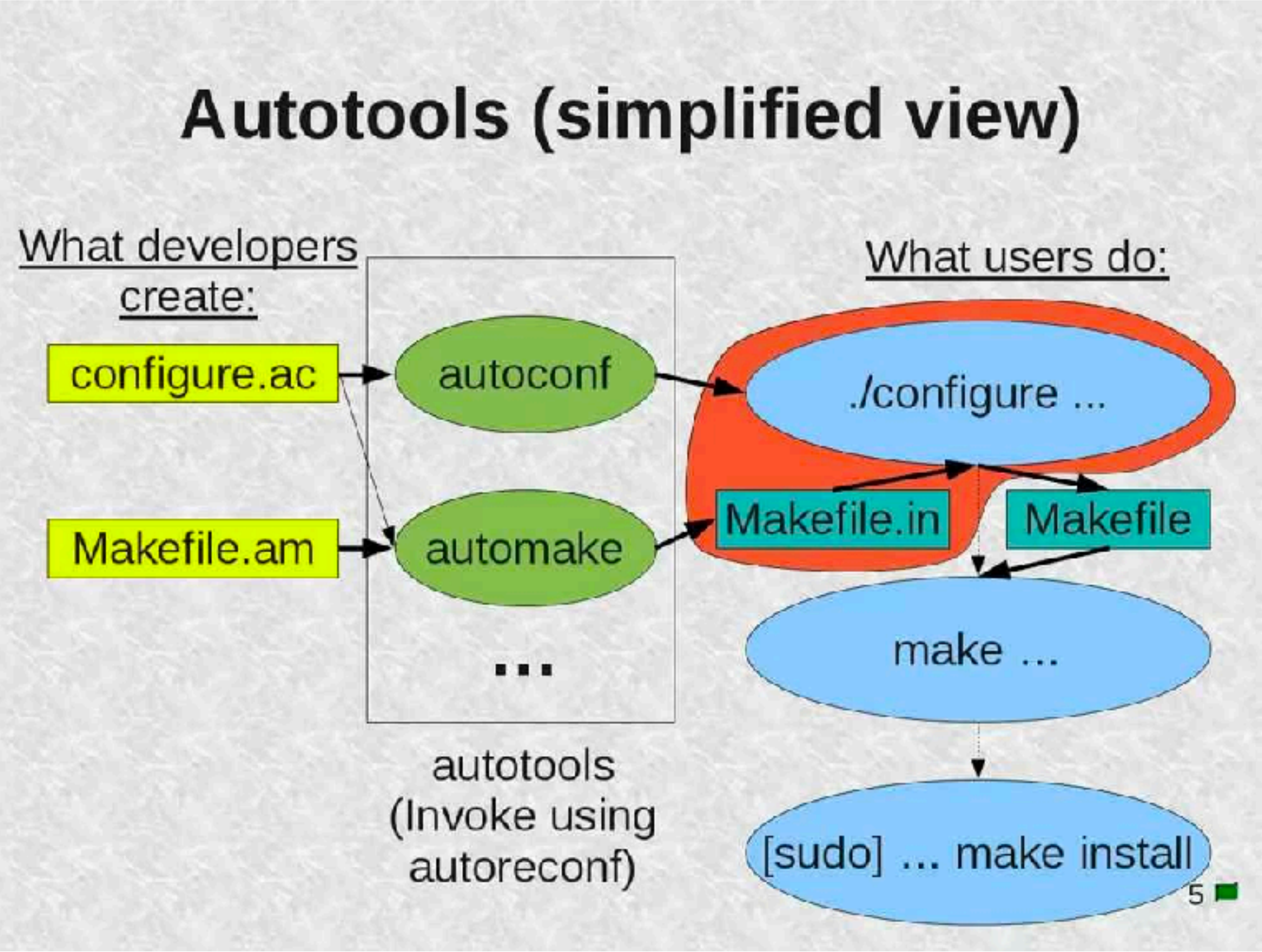
- In shared environments users can still have very specialized workflows and dependencies

- Dependencies are often not useful for all users (e.g.Python)

- Users don’t have rights to install system wide

- Solution: Download source (.tar.gz or tar.bz2) which contain instructions to compile and make your own executable binary and install in your home directory
    - Usually: `./configure; make; make install`

Autotools



Autotools (simplified view)

What developers create:

configure.ac

Makefile.am

autoconf

automake

...

autotools (Invoke using autoreconf)

What users do:

./configure ...

Makefile.in

Makefile

make ...

[sudo] ... make install

5

Installing software with make in your home directory

- Example: the GEOS library
- https://trac.osgeo.org/geos/wiki
- Discussion: https://github.com/phayes/geoPHP/wiki/Geos-installation-on-centos6

```
$ tar -xvjf geos-3.6.2.tar.bz2
$ cd geos-3.6.2/
$ ./configure —help
$ ./configure —prefix=$HOME
$ make
$ make install
```

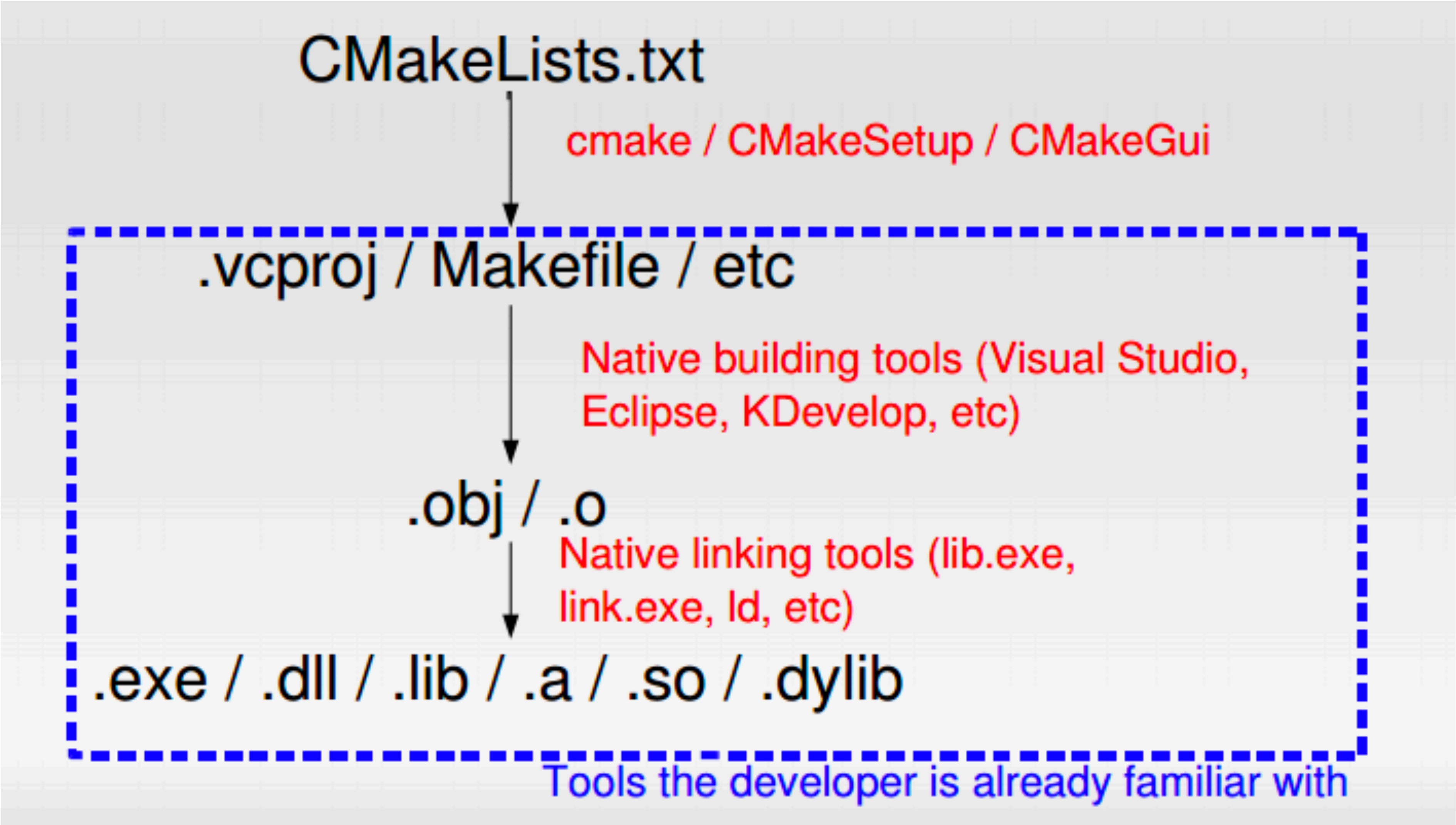
c*make* - a modern version of make

- is a family of tools designed to build, test and package software

- Controls the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces (cross-platform)

- Information for building the project is conveyed to make using a `cmakelists.txt` file

cmake build flow



CMakeLists.txt

cmake / CMakeSetup / CMakeGui

.vcproj / Makefile / etc

Native building tools (Visual Studio, Eclipse, KDevelop, etc)

.obj / .o

Native linking tools (lib.exe, link.exe, ld, etc)

.exe / .dll / .lib / .a / .so / .dylib

Tools the developer is already familiar with

cmake Usage Example 1

- see `CMakeLists.txt in cmake/helloworld`

```
[plindner@opuntia helloworld]$ more CMakeLists.txt
#cmake_minimum_required (VERSION 2.6)
project (helloworld)
add_executable(helloworld helloworld.cpp)
[plindner@opuntia helloworld]$ mkdir build
[plindner@opuntia helloworld]$ cd build/
[plindner@opuntia build]$ cmake ..
-- The C compiler identification is GNU 4.4.7
-- The CXX compiler identification is GNU 4.4.7
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/plindner/intro2linux_make/cmake/
helloworld/build
[plindner@opuntia build]$ make
Scanning dependencies of target helloworld
[100%] Building CXX object CMakeFiles/helloworld.dir/helloworld.cpp.o
Linking CXX executable helloworld
[100%] Built target helloworld
```

Opuntia only!

cmake Usage Example 2

- see `CMakeLists.txt in cmake/helloworld`

```
[plindner@opuntia using_libraries]$ more CMakeLists.txt
cmake_minimum_required (VERSION 2.6)

SET(CMAKE_C_COMPILER /share/apps/intel/composer_xe_2015.3.187/
bin/intel64/icc)
SET(CMAKE_CXX_COMPILER /share/apps/intel/composer_xe_2015.3.187/
bin/intel64/icpc
)

SET(CMAKE_CXX_FLAGS STRING=-openmp)
SET(CMAKE_C_FLAGS STRING=-openmp)

project (svd)

add_executable(svd svd.c)

INCLUDE_DIRECTORIES(/share/apps/intel/mkl/include)
LINK_DIRECTORIES(/share/apps/intel/mkl/lib/intel64)
TARGET_LINK_LIBRARIES(svd mkl_intel_lp64 mkl_core
mkl_intel_thread pthread)
```

Opuntia only!

Further reading

- Gnu make https://www.gnu.org/software/make/manual/

- Cmake https://cmake.org/cmake-tutorial/

Customizing the Environment

Recap Environment Variables

- An environment variable is a shell variable that is exported to make it available in all sub-shells.

- The behavior of the UNIX system is largely determined by the settings of these variables.

- The *set* statement displays a complete list of all variables.

- It's the *env* command (or *export* statement) that shows only the environment variables.

Recap Environment Variables

- Setting a variable to an environment variable in different shells is shown as follows:
  - Bourne shell: x=5; export x
  - C Shell: setenv x 20
  - Korn Shell: export x=5

- PATH is a system variable that contains a colon-delimited list of directories that the shell looks through to locate a command invoked by a user.

Significance of the Environment (System) Variables

- HOME shows your login directory.
- LOGNAME shows your username.
- MAIL shows the mailbox location.
- PS1 stores the primary prompt string. PS2 stores the secondary prompt string.
- CDPATH stores the directory search path.
- SHELL stores the shell you are using.
- TERM indicates the terminal type that is used.

Significance of the Environment (System) Variables

- The bash shell stores the promo information in a couple of variables (PS1 .. PS4, PROMPT_COMMAND)

- The bash shell introduces a history feature that allows users to reexecute previous commands without reentering them.

- Every command in the history list has an event number. The *history* command displays all events.

- The bash shell uses PATH as the command search path.

Aliases

- All shells apart from Bourne support the use of aliases that let you assign shorthand names for frequently used command.

- Examples of using aliases in bash shell are shown in below (must be defined in .bash_rc):

```
alias mydir=‘ls -l’

alias ls=‘ls -Fax`
```

Command History (C Shell and bash)

- The `!` command is used to repeat previous commands in C Shell.
- `!!` repeats previous command.
- `!11` repeats event number 11.
- `!-2` repeats the command before the last one.
- `!v` repeats last command beginning with v.
- `!grep:s/William/Bill` repeats previous grep command with Bill instead of William.
- `^bak^doc` substitutes first instance of bak.

In-Line Command Editing in Korn Shell and bash

- You can perform vi and emacs like in-line editing of the command line by using `set -o vi` or `set -o emacs`.
- Suppose you chose vi. Press `[Esc]` to take you to vi's Command Mode.
- You can use the /pattern sequence.
- Use i, a, A, and so forth to enter the Input Mode.
- Use set +o to turn off in-line editing.
- The default in-line editing in bash is emacs.

Auto Completion

- Korn and bash support a feature called filename completion, which has been enhanced in the modern version of these shells to support.

- Completion of a filename used as an argument to a command.
  - Completion of the command name itself.
  - This means that you may not have to enter the complete command or filename.

Miscellaneous Features

- The ~ acts as a shorthand representation of the home directory.

- `cd ~juliet` effectively becomes `cd $HOME/juliet`.

- We have assigned values to many environment variables, defined aliases and used set options. To make these settings permanent, you'll have to place them in the system's startup scripts.

The Initialization Scripts

- Every shell uses at least one startup script that is placed in the user’s home directory.

- Look in your home directory with *ls –a*, and you’ll find one or more of these files:

  - .profile (Bourne Shell)

  - .login, .cshrc and .logout (C Shell)

  - .profile and .kshrc (Korn Shell)

  - .bash_profile (or .profile or .bash_login), .bashrc and .bash_logout (bash).

The Initialization Scripts

- A script can belong to one of three categories:

- *Login script* – This is a startup script that is executed when a user logs in (.login, .profile and .bash_profile).

- *Environment script* – This file is executed when a sub-shell is run from the login shell. It is often referred to as the rc script (.cshrc, .kshrc and .bashrc).

- *Logout script* – Only the C shell and bash use a logout script (.logout and .bash_logout).

The Initialization Scripts

- There are two commands which run any shell script without creating a sub-shell – the `. (dot)` and `source` command.

- The C shell uses `source`, Bourne and Korn shell use the *dot*, and bash uses both.

- When you log in, you see an interactive shell that present a prompt and waits for your requests.

- When you execute a shell script, you call up a *noninteractive* shell.

The Initialization Scripts

- In the Bourne shell login, the shell executes these two files: `/etc/profile` and `.profile` in user’s home directory.

- In the C shell login, the shell runs three scripts in the order: /etc/login or /etc/.login, ~/.cshrc, and then ~/.login.

- In the Korn shell login, the scripts are executed in this order: /etc/profile, ~/.profile, and then ~/.kshrc.

- In the bash shell login, the scripts are executed in this order: `/etc/profile, ~/.bash_profile, ~/.bash_login, ~/.profile`, and then `~/.bashrc`.

Example .profile

```
plindner@max:~$ vi .profile


#Add GDAL commands

export PATH="/Library/Frameworks/GDAL.framework/Versions/2.1/Programs:$PATH"


#colorful terminal

export PS1="\[\033[36m\]\u\[\033[m\]@\[\033[32m\]\h:\[\033[33;1m\]\w\
[\033[m\]\$ "

export CLICOLOR=1

export LSCOLORS=ExFxBxDxCxegedabagacad

alias ls='ls -GFh'


#meteor

export hnetsftp=myspecialpassword
```

Example .bash_rc

```
[plindner@opuntia ~]$ more .bashrc
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi
# Uncomment the following line if you don't like systemctl's auto-paging feature:
# export SYSTEMD_PAGER=
# User specific aliases and functions
alias cerbero='~/git/cerbero/cerbero-uninstalled'
alias cerbero='/project/cacds/build/gstreamer/git/cerbero/cerbero-uninstalled'
alias cerbero='/project/cacds/build/gstreamer/cerbero/cerbero-uninstalled'
[plindner@opuntia ~]$ more .bash_profile
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
```

Further Reading

- https://www.tutorialspoint.com/unix/unix-environment.htm