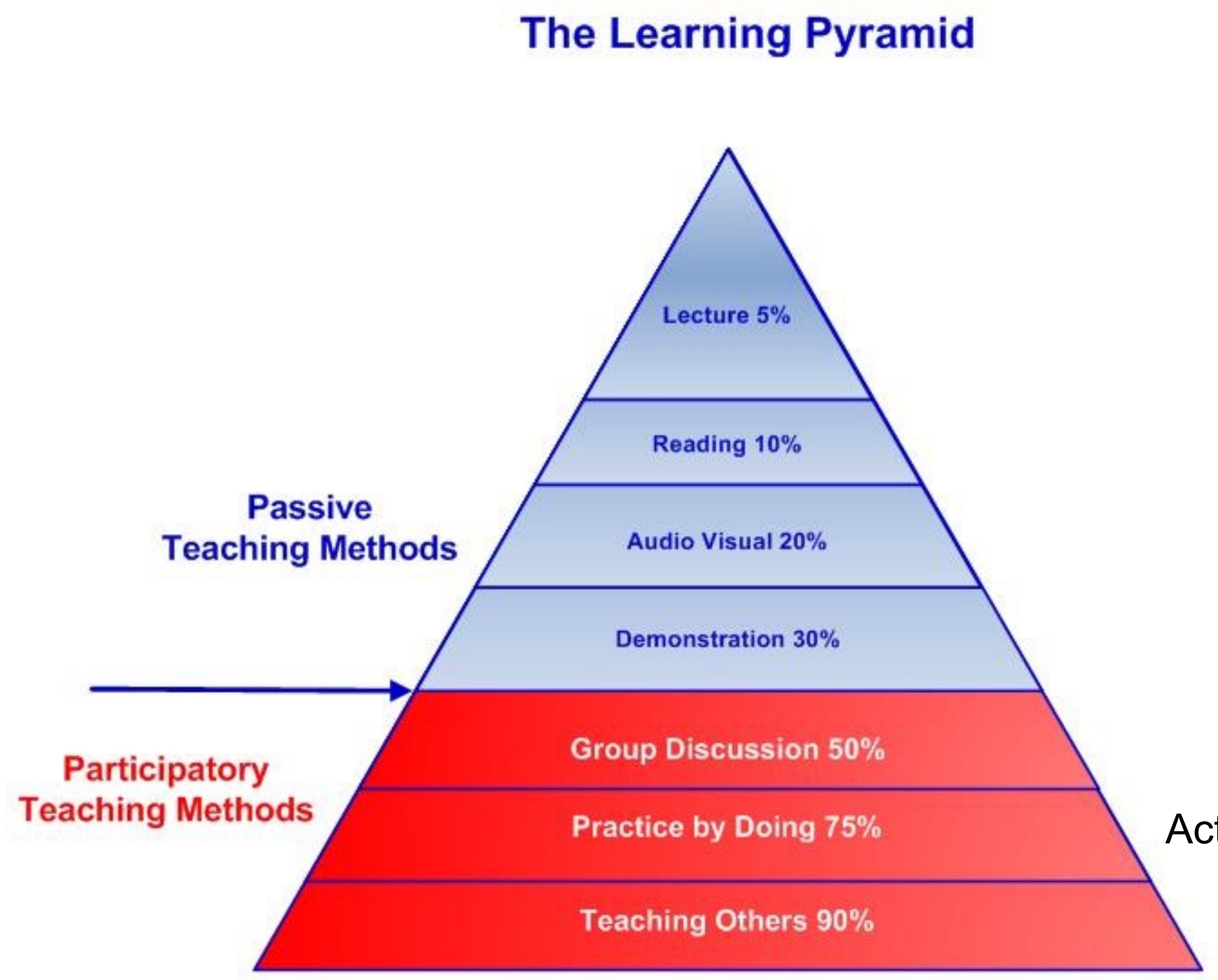
Key skills for using HPC systems

Tim Mattson

Why should we move away from a lecture-dominated format for our course?

Learning Retention Rates for Different Teaching Methods



Active Learning

Peer Coaching

https://thepeakperformancecenter.com/educational-learning/learning/principles-of-learning/learning-pyramid/retention-rates/

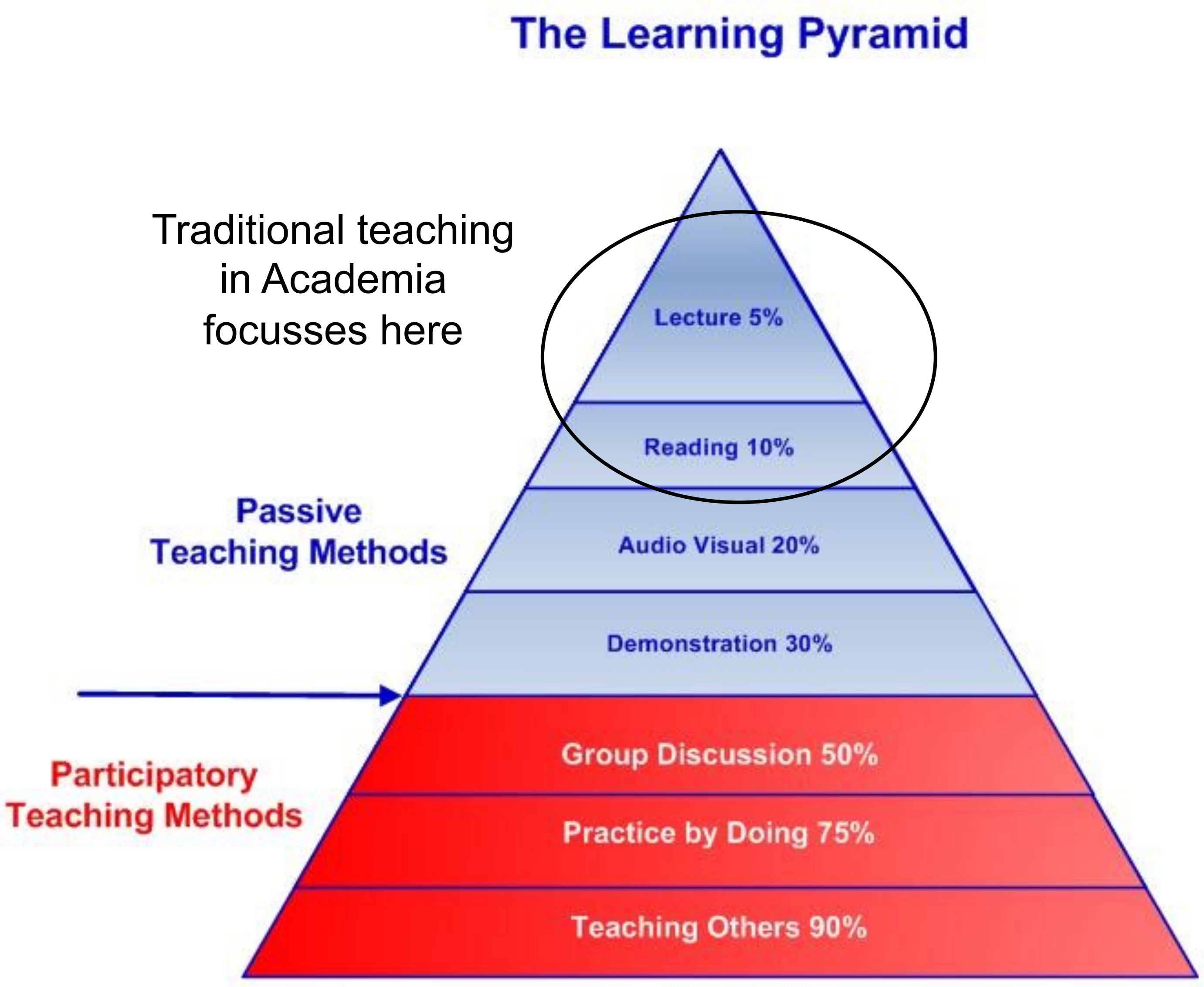
How to become an educator

- To teach programming:
  - Achieve mastery … a Ph.D. helps
  - Agree/volunteer to teach.

I started teaching OpenMP in 1998

OpenMP™

Learning Retention Rates for Different Teaching Methods



https://thepeakperformancecenter.com/educational-learning/learning/principles-of-learning/learning-pyramid/retention-rates/

I became a kayak instructor around 2000

Photo © by Greg Clopton, 2014

I am a recently retired kayak coach:

- **Level 5 coastal kayaking (advanced open ocean)**
- **Traditional kayaking endorsement**
- **Level 3 white water(basic white-water kayaking)**

and Kayak Instructor Educator (level 3 coastal kayak)

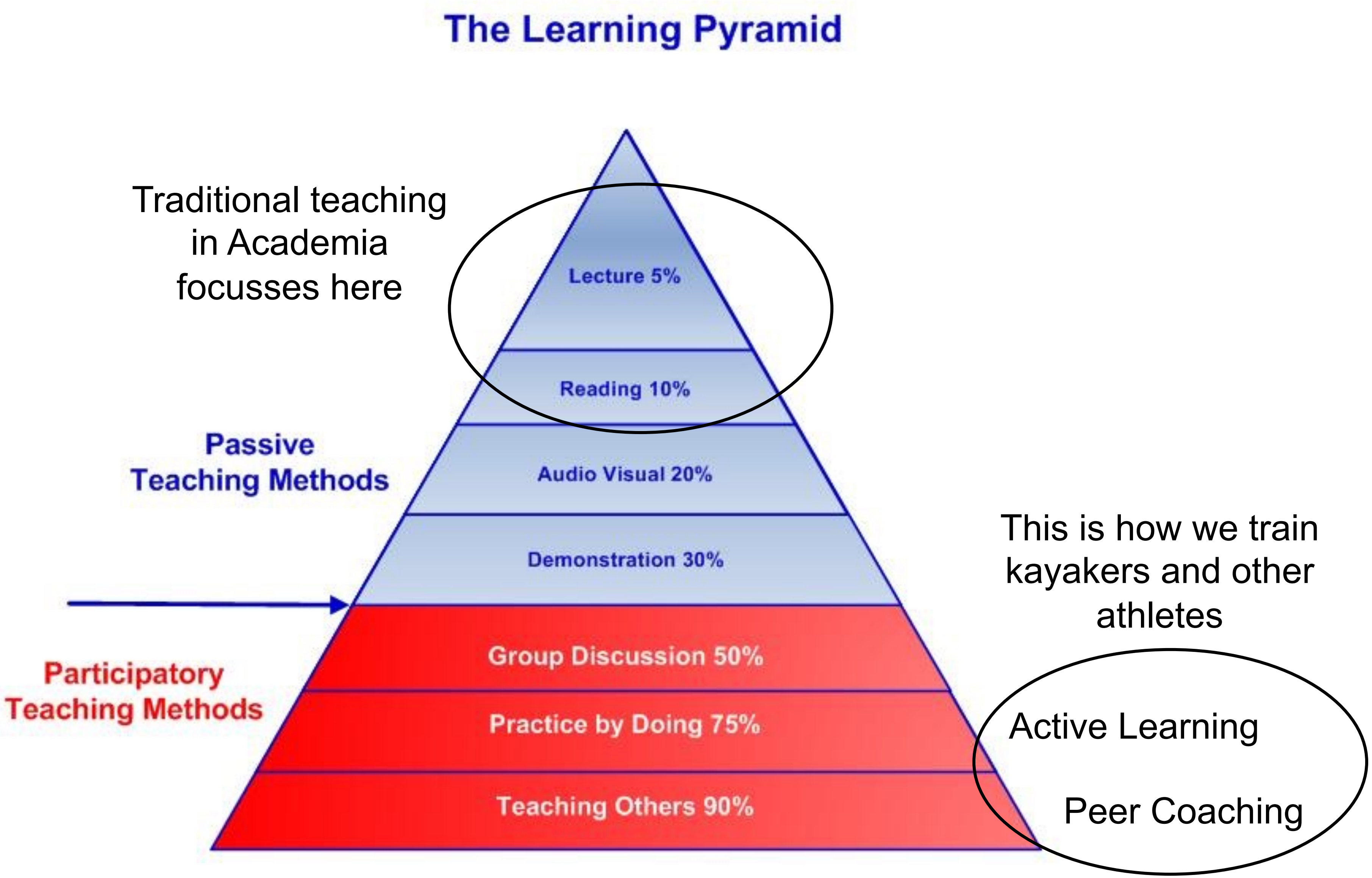
How to become an educator

- To teach programming:
  - Achieve mastery … a Ph.D. helps
  - Agree/volunteer to teach.
- To teach kayaking*
  - Achieve mastery in kayaking
  - Complete a three-day intensive workshop on learning theory and coaching technique.
  - Video stroke-analysis to document shortcomings in paddling technique.
  - Additional training to resolve all technical shortcomings (Usually a year or more of work).
  - Work with a mentor to refine teaching technique.
  - Attain advanced first aid certification and refine rescue skills
  - Pass a video exam of core techniques.
  - Pass a three-day certification exam.
  - Repeat for each level of instructor certification (levels 1 to 5) and each discipline you wish to teach (white water, coastal, surf, etc).

ACA
CANOE · KAYAK · SUP · RAFT · RESCUE

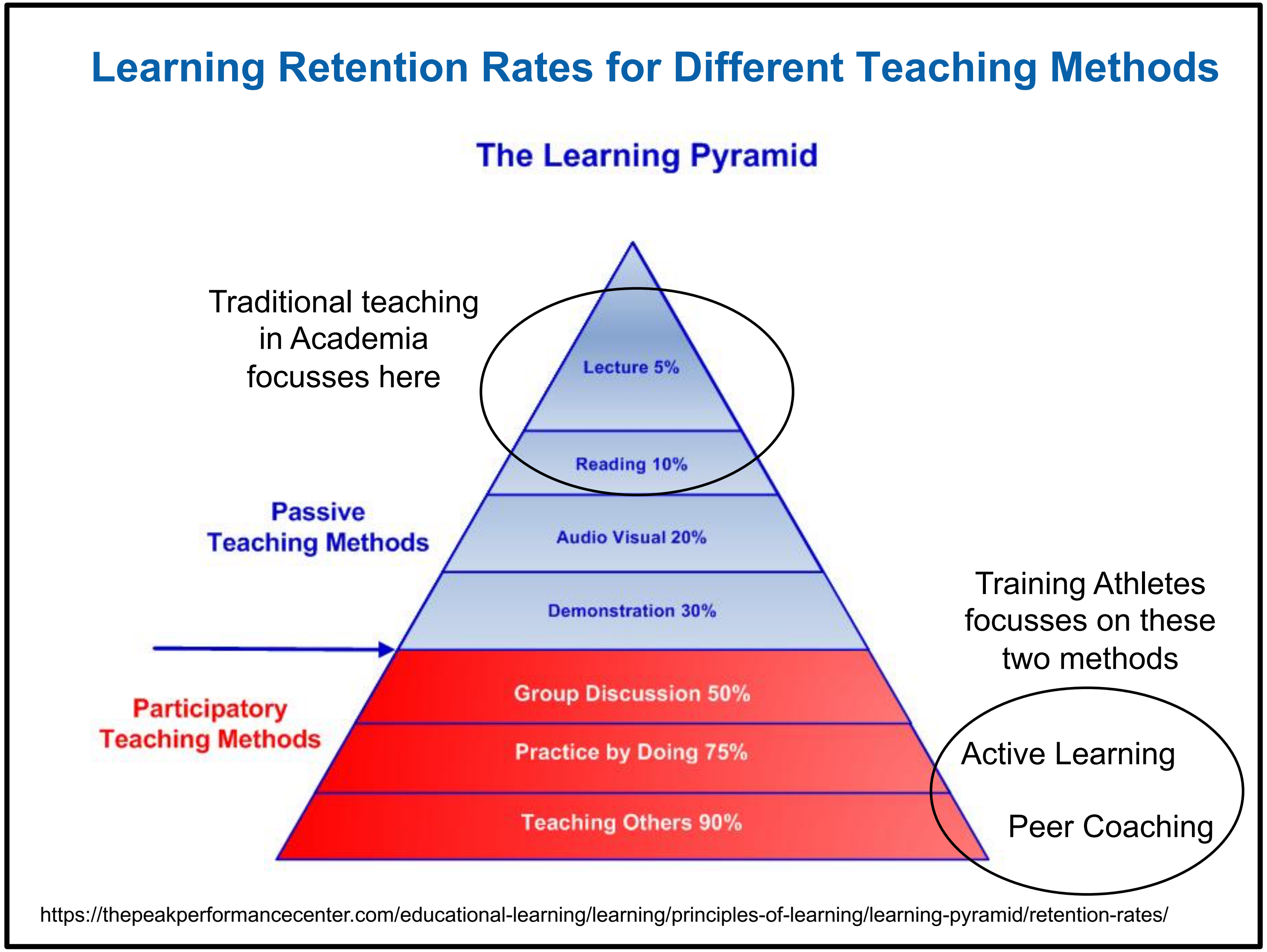
*This process is used by the American Canoe Association to certify kayak instructors

Learning Retention Rates for Different Teaching Methods



https://thepeakperformancecenter.com/educational-learning/learning/principles-of-learning/learning-pyramid/retention-rates/

We know its better … why don’t we teach programmers the same way we teach kayakers?



So ... As we focus on programming in HPC, we are going to shift to *active learning* in which you will learn by writing, debugging, and analyzing code

Topics

OpenMP is the most commonly used parallel programing model today.

It is also the easiest to use … we can cover most topics in parallel programing theory using OpenMP

In hands-on sessions we will be writing code together in class.

Bring well-charged laptops.

Mon	Tue	Wed	Thu	Fri	Sat
Intro to Sci Comp					
Computer Architecture				**Using HPC systems + start OpenMP**	**OpenMP**
OpenMP					
Super Computing 2024 in Atlanta Georgia					
OpenMP					**GPU programming**
GPU Programming				**Cluster computing with MPI**	
The joys of computer arithmetic				**How databases work .. Student presentations**	
The Future of computing … Student presentation					

Supporting our transition to hands-on learning

- Local systems (including your laptop) and how we will use them

- The Linux Operating System (OS) for HPC programmers

- Basic tools for HPC software developers working with Linux

- C … the high-level assembly code of computing. The simple, minimal subset.

Modern Supercomputers are accessed remotely. Often you need to work with them through a command line prompt … so you need to know the basics of Linux.

There are a core set of tools used on supercomputers to manage software and edit code.

We need a programming language that makes these computers visible. That means C or C++ … we'll pick C since its easier to learn.

Your homework from last lecture …

Starting with our next lecture, we will shift to hands on learning

- People learn by doing. The next time we get together we'll talk about our transition to hands-on learning.
- This means learning about Linux, the C programming language, working from a command line prompt and more.
- To prepare I want each of you to make sure your laptop has an actual Gnu compiler installed by our next time together.
- For information on how to accomplish this … see these useful guides, the first in English and the second covering the same material in Italian.
  https://github.com/giacomini/ebernburg2024/tree/main/guides
  https://github.com/Programmazione-per-la-Fisica/howto/tree/main/other-OSes

Note: these come from Francesco Giacomini at the University of Bologna.   His course emphasized C++.   We will focus on C. And he advises VSCode. That's OK, but I suggest vi (or vim) instead.

Your homework from last lecture …

Did you all do this?

Starting with our next lecture, we will shift to hands on learning

- People learn by doing. The next time we get together we'll talk about our transition to hands-on learning.
- This means learning about Linux, the C programming language, working from a command line prompt and more.
- To prepare I want each of you to make sure your laptop has an actual Gnu compiler installed by our next time together.
- For information on how to accomplish this … see these useful guides, the first in English and the second covering the same material in Italian.
  https://github.com/giacomini/ebernburg2024/tre
  https://github.com/Programmazione-per-la-Fisi

Note: these come from Francesco Giacomini at t
C++. We will focus on C. And he advises VSCo

Run the command gcc –v. Which version of the gcc compiler are you using?

If you are on Apple system, you will need to use gcc-14 or some other Gnu compiler since gcc redirects to Apple's own clang compiler.

Systems available for this course

- Two options for doing exercises:
  1. Use your own laptop … and excellent choice for OpenMP and workable but non-ideal choice for MPI
  2. Use systems in the INFN cloud … best choice for MPI and only choice for GPU

- We are starting with OpenMP so hopefully we can work with our laptops for now which will give me a bit more time to setup the systems in the cloud (which are essential for us on Nov. 30 when we start on GPU programming).

- Note: an added bonus to doing OpenMP on your own laptops is then you have an environment you can work with and keep practicing on your own after the course is over.

- You can also load MPI on your laptop, though I hope we have an actual cloud-based cluster ready by the time we get to MPI.

Use homebrew to install gnu compilers on your Apple laptop

I tested this on a new (July 2023) MacBook Air with an Apple M2 CPU

Warning: by default Xcode uses the name gcc for Apple's clang compiler.
Use Homebrew to load a real, gcc compiler.

- Go to the homebrew web site (brew.sh). Cut and paste the command near the top of the page to install homebrew (in /opt/homebrew):

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

- Add /opt/homebrew/bin to your path. I did this by adding the following line to .zshrc

```
% export PATH=/opt/homebrew/bin:$PATH
```

- Install the latest gcc compiler

```
% brew install gcc
```

- This will install the compiler in /opt/homebrew/bin. Check /opt/homebrew/bin to see which gcc compiler was installed. In my case, it installed gcc-13
- Test the compiler (and the openmp option) with a simple hello world program

```
% gcc-13 –fopenmp hello.c
```

Gnu Compilers on Apple Laptops: MacPorts

I have not tested this in a long time.
I greatly prefer homebrew.

But if you prefer MacPorts, this procedure should work.

- To use a Gnu compiler on your Apple laptop:
- Download Xcode. Be sure to choose the command line tools that match our OS.
- Download and use MacPorts to install the latest gnu compilers.

```
sudo port selfupdate
```

Update to latest version of MacPorts

```
sudo port install gcc13
```

Grab version 13 gnu compilers (5-10 mins)

```
port select --list gcc
```

List versions of gcc on your system

```
sudo port select --set gcc mp-gcc13
```

Select the mp enabled version of the most recent gcc release

```
gcc -fopenmp hello.c
```

Test the installation with a simple program

Supporting our transition to hands-on learning

- Local systems (including your laptop) and how we will use them
- The Linux Operating System (OS) for HPC programmers
- Basic tools for HPC software developers working with Linux
- C … the high-level assembly code of computing. The simple, minimal subset.

Linux: HPC's Ubiquitous Operating System (OS)

Tux … the official mascot of Linux

- Unix comes from Bell Labs in the late 1960's and 70's. It was mostly written in C.
  - The new idea was an OS as a set of distinct services layered around a compact kernel
  - It is organized around a file-system … leading to the saying ***Everything is a file***.
- Bell Labs licensed Unix to a number of commercial and academic users in the late 1970's resulting in a number of Unix variants:
  - University of California Berkely → The BSD open source OS
  - Sun microsystems → SunOS
  - Hewlett Packard → HP-UX
  - IBM → AIX
- Today, three* commonly encountered operating systems survive from the Unix lineage
  - Apple's Unix core based on BSD called Darwin → run through the terminal sessions on an Apple laptop.
  - The Linux operating system → dominant in servers, HPC clusters, and cloud environments on individual nodes
  - Microsoft's Windows Subsystem for Linux (WSL) → run through terminal session on Windows systems
- The three operating systems vary internally, but to the user they present common interfaces.

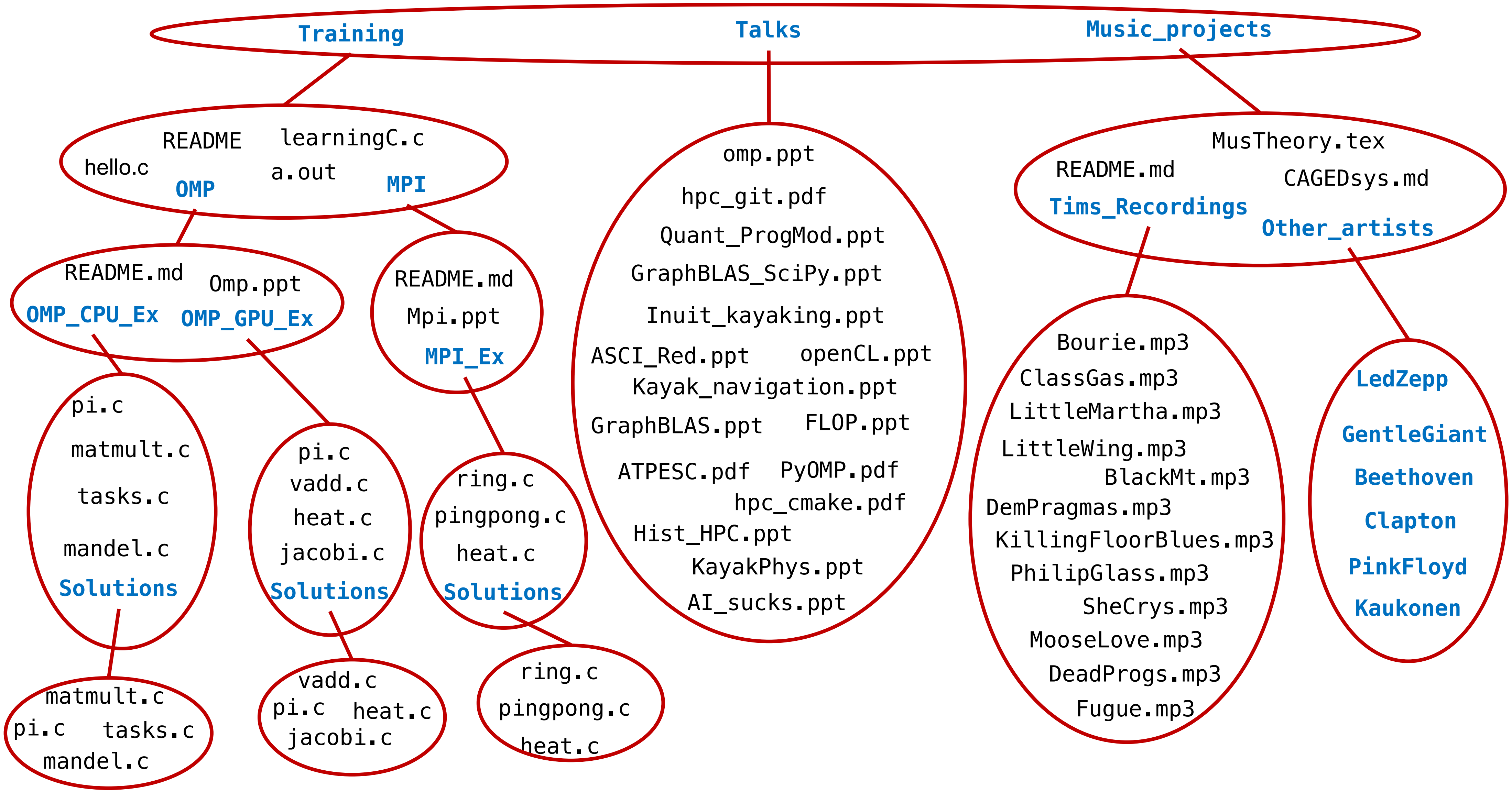
If you work in High Performance Computing, you will need to be comfortable with the most common Linux commands

*Actually four, given that Android is based on Linux.

Tux source: lewing@isc.tamu.edu Larry Ewing and The GIMP

Linux file system

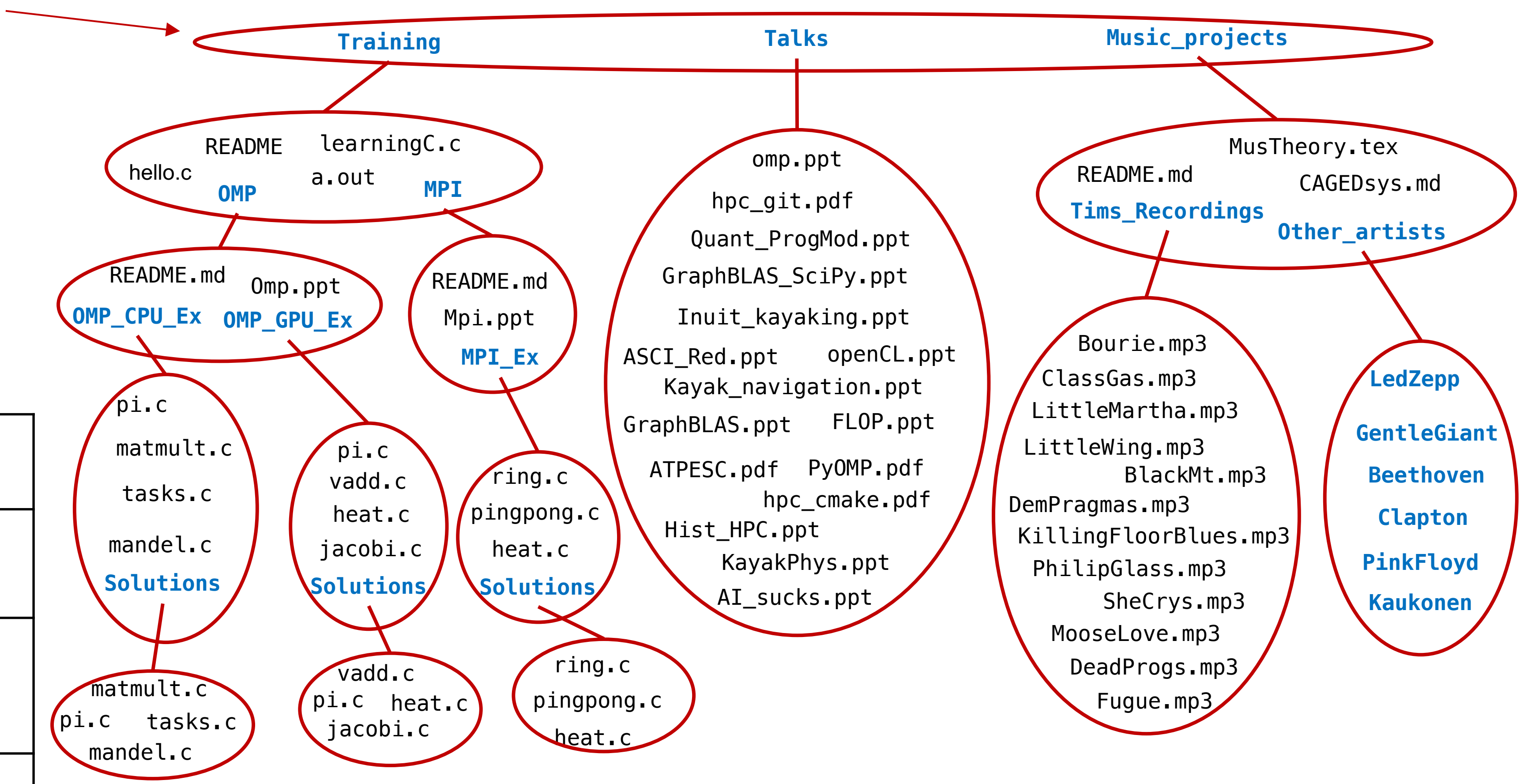
A file system is a tree of files that represent collections of files, directories ( ), **links to directories**, and plain files



Linux: Directories

At login, you are in the home directory.

% ssh tgmattso@sys.nersc.gov
passwd:
Logged on sys.nersc.gov
% pwd
/home/tgmattso



Commands for working with directories.

pwd	Print working directory
cd foo	Change working directory to foo
ls	list files in current director. ls –a to see hidden files, ls –l to see information about the files
mkdir foo	Create a directory named foo
rmdir foo	Remove the directory named foo

Linux: files within directories

Change directories with the **cd** command

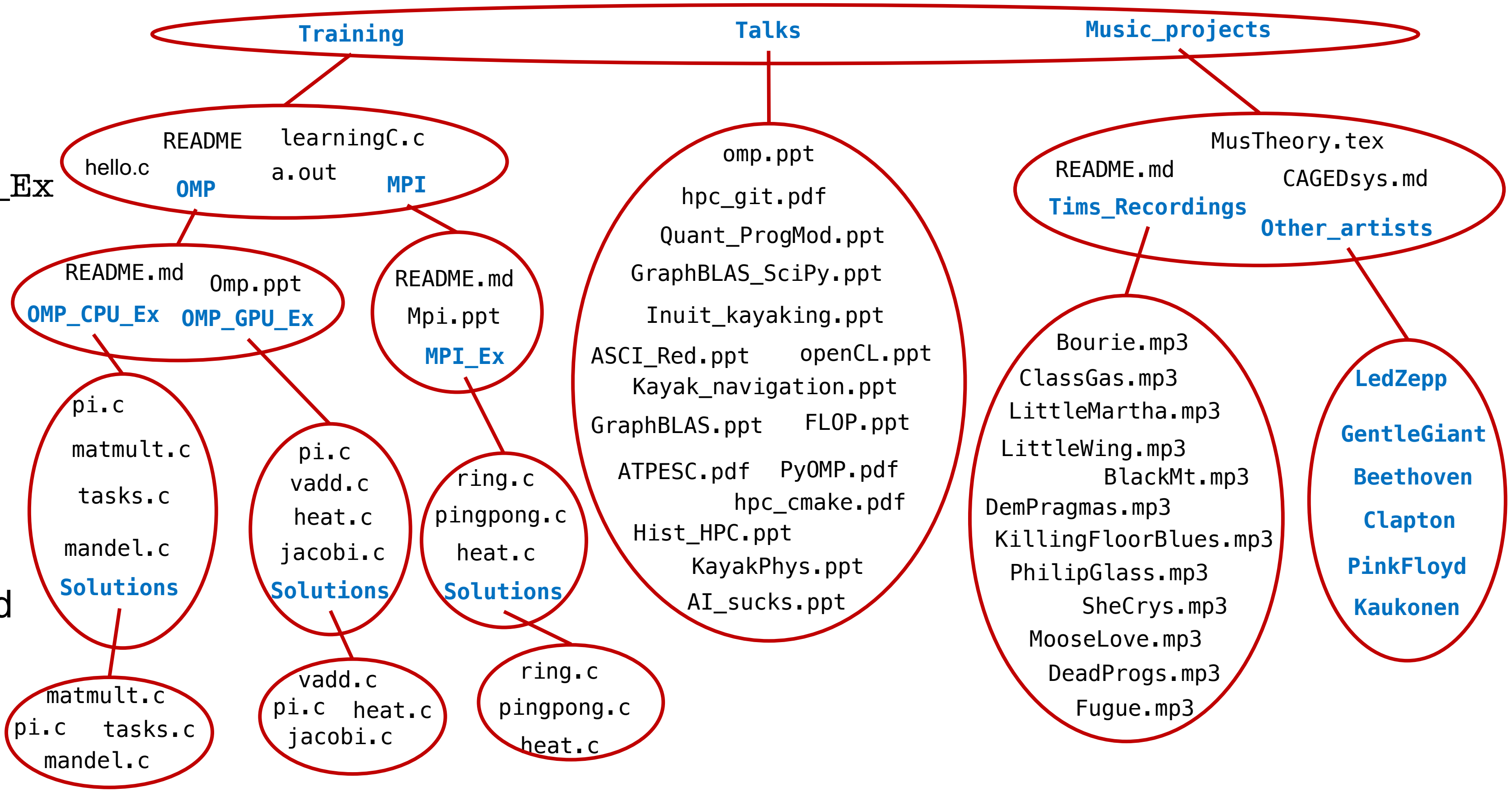
% cd Training
% cd OMP/OMP_ CPU_Ex
% pwd
/Users/tgmattso/Training/OMP/OMP_CPU_Ex

The **ls** command shows directory contents

% ls
mandel.c   tasks.c
matmult.c   Solutions
pi.c

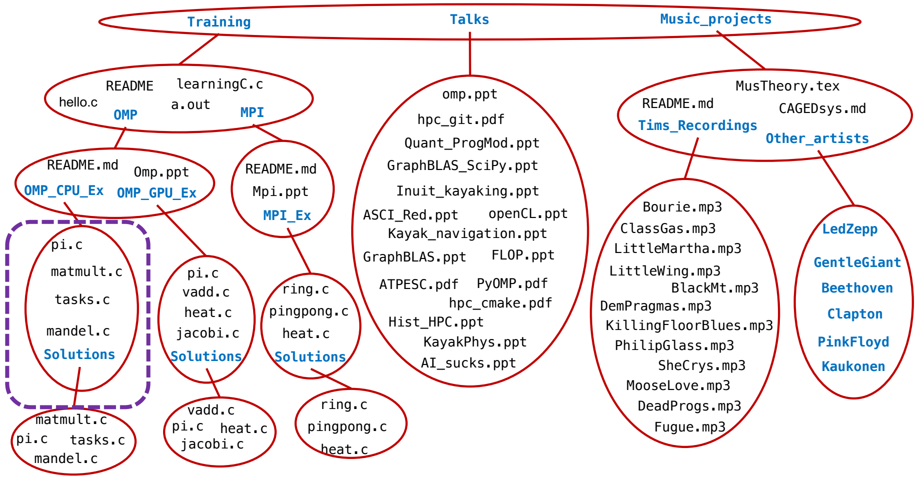
The path to a file lists directories separated by a slash (**/**). There are multiple ways to reference a particular file.

% ls ../../MPI
Mpi.ppt
MPI_Ex
README.md
% ls ~/Training/MPI
Mpi.ppt
MPI_Ex
README.md



.	The current directory
..	The parent directory
~	The home directory

Linux file system: working with files



Let's stay in ~/ Training/OMP/OMP_CPU_Ex for now

```
% pwd
/Users/tgmattso/Training/OMP/OMP_CPU_Ex

% ls
mandel.c        tasks.c
matmult.c       Solutions
pi.c
```

The **cat** command concatenates the contents of a file to the standard output (by default, your screen).
This may overflow a screen, so we can see one screen at a time with the **more** command.

For now let's just look at the first 3 lines of a file

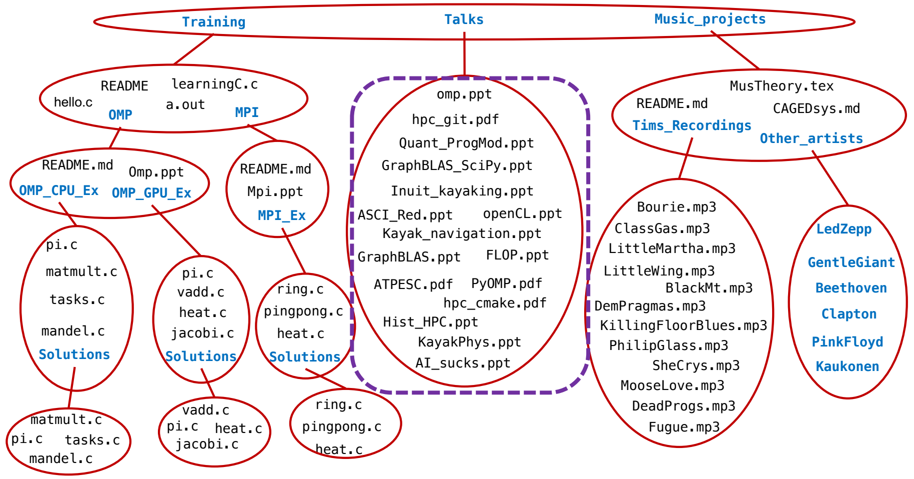
```
% head –n 3 pi.c
//
//
// This program will numerically compute the integral of
```

We can move, copy and remove files

```
% cp pi.c pi2.c
% mv pi.c  poo.c
% rm tasks.c mandel.c matmult.c
% ls p*
pi2.c
poo.c
```

cat pi,c	Dump contents of pi.c to standard output
more pi.c	Display pi.c one screen at a time
head pi.c	Display the first 20 lines of pi.c or fewer with -n
cp f1 g2	Create a copy of f1 named g2
mv f1 g2	More or rename file f1 to g2
rm f1	Remove (delete) the file named f1
tail pi.c	Display the last 20 lines of pi.c or fewer with -n
*	Wildcard for matching files. **rm *.c** deletes all files that end with .c. **ls *o*** shows all file names that have an o

Linux: working with linux commands



Let's move to **/home/tgmattso/Talks** for now

```
% cd ~/Talks
% ls Inuit*
Inuit_kayaking.c
```

We can see all the commands we've done so far with the **history** command

```
% history
1001  cd Training
1002  cd OMP/OMP_ CPU_Ex
1003  pwd
1004  ls
1005  ls ../../MPI
1006  ls ~/Training/MI
1007  pwd
1008  ls
1009  head -n 3 pi.c
1010  cp pi.c pi2.c
1011.  mv pi.c poo.c
1012  rm tasks.c mandel.c matmult.c
1013  ls p*
1014  cd ~/Talks
1015  ls Inuit *
1016  history
```

The system remembers past commands and displays the last 15.

We can use past commands to generate new commands

!!	Repeat last command
!p	Repeat last command that started with the letter p
!1008	Repeat command number 1008
!$	Replace with the last string on the previous line

If you start typing characters and hit tab, it will complete the string based on matching files in the current directory. You can also use the arrow keys to edit a command

% ls Qu **Then hit the tab key to get the full name**
Quant_ProgMod.ppt
% rm !$ **Remove the file Quant_ProgMod.ppt**

On my Apple laptop, the arrow keys let you move up to past commands and then → and ← to move a cursor and edit a command.

Linux: Command line interpreters or shells

The default shell for MacOS

- Commands are interpreted by a command line interpreter (shell). There are a few common shells: **bash**, **csh**, **zsh**
- They largely act the same … customize your shell at login through a file: **.bashrc**, **.cshrc**, **.zshrc**
  - Files that start with a period are hidden. An **ls** command won't show them. To see them use **ls –a**

- When you type a command, the shell searches a path to find that command. Your path is defined by an environment variable **PATH**. To see the value of your **PATH**, use **$** to get the value and **echo** it to the standard output.

```
% echo $PATH
/Users/tgmattso/anaconda3/bin:/Users/tgmattso/anaconda3/condabin:/opt/homebrew/bin:/usr/local/bin:/System/Cryptexes/A
pp/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin:/var/run/com.apple.security.cryptexd/codex.system/bootstrap/usr/local/bin:/v
ar/run/com.apple.security.cryptexd/codex.system/bootstrap/usr/bin:/var/run/com.apple.security.cryptexd/codex.system/b
ootstrap/usr/appleinternal/bin:/Library/Apple/usr/bin:/Library/TeX/texbin
```

- You often need to create and set environment variables when working with software. For example, in OpenMP we can set the default number of threads for a program to use.

```
% export OMP_NUM_THREADS=5
% printenv
```

This command will print all the defined environment variables.
(It can be very long so we won't show the output here)

- There are some special characters used with commands. These include >, <, |, >>

```
% ls > listFile
% ./myprog < data
% grep "foo" listFile | wc
% ls >> listFile
```

Execute **ls** and direct output to the file **listFile** … overwriting contents if the file already exists

Run **myprog** located in the current directory (**./**) taking input from the file **data**

Run **grep** on listFile to find lines with the string "foo". Pipe the output to the command **wc** (word count)

Execute **ls** and append the output to the end of the file listFile

Linux: An example .zshrc file

- The following is the .zshrc file from my laptop. The analogous files for other shells are similar and can be found online.

```
% cat ~/.zshrc
export PATH=/opt/homebrew/bin:$PATH

# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/Users/tgmattso/anaconda3/bin/conda' 'shell.zsh' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/Users/tgmattso/anaconda3/etc/profile.d/conda.sh" ]; then
        . "/Users/tgmattso/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/Users/tgmattso/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<
```

It's not uncommon to explicitly update your path

All of this was inserted into the file by conda during installation of the system on my laptop. I didn't add any of this myself

- The .zshrc file is an example of a **shell script** … this is a collection of commands as you'd type into a command line placed in a file. You can do this to build custom commands or to submit work to be carried out to a batch queue scheduler

I wrote the Shell script to save how I built and ran the program

```
% cat pimc.zsh
#!/bin/zsh
count=$1
echo "pimc with $count samples"
echo "Hostname: $(hostname)"
gcc-13 -fopenmp -O3 pimc.c random.c
./a.out $count
```

Command line argument for number of samples

```
% chmod +x pimc.zsh
% ./pimc.zsh 1000
pimc with 100000 samples
Hostname: Tims-MacBook-Air-3.local
 pi mc with 100000 trials
 100000 trials, pi is 3.139920  in 0.002734 seconds
```

chmod on the shell script to tell the system it is an executable (contains commands to run)

Exercise: Working with the Linux Command Line

- Using the features of Linux we have discussed so far, do the following:
  1. Create a file with dozens of lines. Do it using some combinations of Linux commands (not with a text editor).
  2. Using commands strung together **<u>on a single command line</u>**, print lines 4 though 7 of that file.
  3. Write a shell script to accomplish the task in item 2

pwd	Print working directory
cd foo	Change working directory to foo
ls	list files in current director. ls –a to see hidden files, ls –l to see information about the files
mkdir foo	Create a directory named foo
rmdir foo	Remove the directory named foo

grep	Finds line that match a string (and much more)
wc	Returns lines, words, bytes in a file
chmod	Change the read/write/execute mode of a file
printenv	Print to standard out (stdout) all environment variables.
export	Set and environment var …. % export var=value
Echo $var	Print value of var to stdout

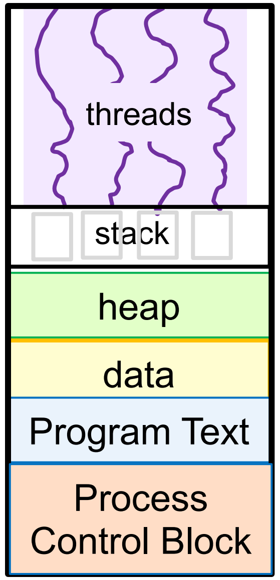
cat pi,c	Dump contents of pi.c to standard output
more pi.c	Display pi.c one screen at a time
head pi.c	Display the first 20 lines of pi.c or fewer with -n
cp f1 g2	Create a copy of f1 named g2
mv f1 g2	More or rename file f1 to g2
rm f1	Remove (delete) the file named f1
tail pi.c	Display the last 20 lines of pi.c or fewer with -n

.	The current directory
..	The parent directory
~	The home directory

```
% ls > listFile
% ./myprog < data
% grep "foo" listFile | wc
% ls >> listFile
```

Linux: working with processes

- A process is an instance of a program managed by the operating system.
- At a high level, we can think of a process as consisting of the following components
  - Process control block (ID, state, plus I/O and other OS managed resources for the process)
  - One or more threads that execute the program
  - The program text (that is the executable machine code for the program).
  - The static data region available to all threads (i.e., values known before the program begins execution).
  - Memory as a heap. This is shared memory visible to all threads in the process.
  - Memory managed as a stack local to each block/function/thread



Components of a process

- A process can run in the foreground or in the background

```
% ./pi_mc 100000000 > outputFG
```

Foreground: program completes before you get a prompt for the next command

```
% ./pi_mc 100000000 > outputBG &
```

Background: Command returns so you can run other commands right away

```
% ps
  PID TTY           TIME CMD
96419 ttys001    0:00.06 -zsh
42573 ttys002    0:00.87 -zsh
98015 ttys002    0:00.75 ./pi_mc
```

ps: process status. Lists processes, their ID, the output screen they are attached to, and the CPU time they've used. I had two terminal sessions on my laptop so there were two active **zsh** processes.

- How to stop a bad process.
  - For Foreground (interactive) sessions, hold the control key and press c (control-C) to interrupt the process.
  - For Foreground (interactive) sessions, hold the control key and press c (control-D) to kill the process.
  - For Background sessions, find the process ID with ps and use kill -9

```
% kill -9 98015
```

Learning Details of Linux commands: man

The **man** command will give you the manual page for any Linux command (lots of detail so they pipe it through **more**)

% man ls

```
LS(1)                       General Commands Manual                       LS(1)

NAME
     ls – list directory contents

SYNOPSIS
     ls [-@ABCFGHILOPRSTUWabcdefghiklmnopqrstuvwxy1%,] [--color=when] [-D format] [file ...]

DESCRIPTION
     For each operand that names a file of a type other than directory, ls displays its name as well as any
     requested, associated information.  For each operand that names a file of type directory, ls displays
     the names of files contained within that directory, as well as any requested, associated information.

     If no operands are given, the contents of the current directory are displayed.  If more than one
     operand is given, non-directory operands are displayed first; directory and non-directory operands are
     sorted separately and in lexicographical order.

     The following options are available:

     -@      Display extended attribute keys and sizes in long (-l) output.


     -A      Include directory entries whose names begin with a dot (‘.’) except for . and ...
             Automatically set for the super-user unless -I is specified.
:
```

Space bar for the next page or **q** when you've had enough.

Supporting our transition to hands-on learning

- Local systems (including your laptop) and how we will use them
- The Linux Operating System (OS) for HPC programmers
- Basic tools for HPC software developers working with Linux
- C … the high-level assembly code of computing. The simple, minimal subset.

Editing Text when all you have is a command-line prompt (vi)

- An integrated development environment (IDE) such as VSCode is nice, but sometimes all we have is a command-line prompt.
- This happens often in HPC when we work with remote servers that only give us a command-line prompt.
- The ubiquitous editor is **vi** (or its modern equivalent, **vim**).

```
% vi pi_mc.c
```

Open a **vi** session with the file pi_mc.c, or create a new file with that name.

- The **vi** command opens or creates the indicated file. It takes over the terminal session presenting one with a blank screen.
- Two modes in **vi**:
  - **Command mode**: used to move around the screen, edit the contents, manage files, and enter input mode.
  - **Input mode**: Add text to the file.
- The vi session starts in command mode. The escape key at any point puts you into command mode. Commands are generally organized around a cursor that appears in the text being edited.
- Moving the cursor: Arrow keys or the letters **h** (←), **j** (down), **k** (up), **l** (→)

These commands put you in input mode	
a	Input text after the cursor
i	input text before the cursor
o	Open a new line below the cursor. Input text
O	Open a new line above the cursor. Input text
r	Replace a character
R	Replace text up to the end of the line

dd	Delete line cursor is on. Put in a buffer
yy	Yank the line (put in a buffer, no delete)
dw	delete word. Put in a buffer
d$	Delete cursor to end of line. Put in buffer
x	Delete character. Put in buffer
p	Put buffer after the cursor
u	Undue last command
.	Repeat last command
/lkj	Find characters **lkj,** put the cursor there

Hit colon in command mode to get a prompt	
a	**a** is a number. Got to that line
w	Write vi session to the file
q	Quit vi, return to Linux prompt
wq	Write then quit
q!	Quit without saving
Set nu	Show line numbers
a,b **m** c	Move lines **a** to **b** after line **c**
a	Go to line a. Use $ for last line

Building software with Make

- For our exercises, we build programs spread between 1 or 2 files. We just type a command to compile and link our program (in this case, using –o to indicate that the executable will be named pi):
  % gcc –o pi pi_mc.c random.c
- For more complicated programs, we put the build instructions in a file named makefile and use make:
  % make pi
- How do you create a new makefile? You don't. You take one that works and modify it.
  % cat makefile

See next page

Oh wait … just a few points about compiler flags (the –c and –o compiler flags). You compile code into object files. The linker combines these with libraires at "link time" to produce the executable

% gcc –c Myprog.c ← Just generate the object file which will late be used to make an executable
% gcc –o execName Myprog.o ← name the executable execName

Some other flags you may need
-std=c11 ← use the C'2011 standard for the C code
-fopenmp ← the code is using the OpenMP language. Also include OpenMP libraries at link time
-lm. ← include the standard math library at link time.

A simple makefile

```
#   make       ... to build the program
#   make clean ... remove executables and object files

CC          = gcc-14
CLINKER     = $(CC)
LIBS        = -lm
CFLAGS      = -fopenmp
OPTFLAGS    = -std=c11 -fopenmp -O3

EXES=hello pi_mc

PMC_OBJS  = pi_mc.o random.o

all: $(EXES)

hello: hello.o
	$(CLINKER) $(OPTFLAGS) -o hello hello.o $(LIBS)

pi_mc: $(PMC_OBJS) random.h
	$(CLINKER) $(OPTFLAGS) -o pi_mc $(PMC_OBJS) $(LIBS)

clean:
	rm $(EXES) *.o

.SUFFIXES:
.SUFFIXES: .c .o

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

A simple makefile

```
#   make        ... to build the program
#   make clean ... remove executables and object files
```

Comments defining externally exposed targets. The default case builds all the executables in this makefile and a second target clean

```
CC          = gcc-14
CLINKER     = $(CC)
LIBS        = -lm
CFLAGS      = -fopenmp
OPTFLAGS    = -std=c11 -fopenmp -O3
```

Use variables for things you change often such as compilers, linkers, and flags used in compilation and linking.

```
EXES=hello pi_mc
```

A variable set to the names of all the executables you can build with this makefile

```
PMC_OBJS  = pi_mc.o random.o
```

A variable that lists the object files needed by more complicated programs

```
all: $(EXES)
```

This tells make by default to build all the targets in the variable EXES.

```
hello: hello.o
	$(CLINKER) $(OPTFLAGS) -o hello hello.o $(LIBS)

pi_mc: $(PMC_OBJS) random.h
	$(CLINKER) $(OPTFLAGS) -o pi_mc $(PMC_OBJS) $(LIBS)
```

These are the executables (the targets) we will build. The format is:

Target: dependencies
<<< tab >>> Command to build target

```
clean:
	rm $(EXES) *.o
```

This target has no dependencies and just executes the indicated command

```
.SUFFIXES:
.SUFFIXES: .c .o

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

These are the rules to deal with files that have different suffices. Here we say “add to the default set” a new rule to build a .o file from a .c file

These spaces are tabs

Make summary

- Make is a powerful tool.   It can greatly simplify your life when building complex programs. Plus it is stable and available ”every where”.

- Make documents how you built a program which is important should you ever need to reproduce your results

- Writing makefiles can be painful. The error messages you get are challenging to work with.   This is why most of us DO NOT write makefiles from scratch. We take one that works and modify it to do what we need

A newer tool called cmake is gaining in popularity. It can do much more than make and is therefore preferred by people building commercial-grade applications. But it is much more complicated to use, has APIs that change between versions, and is not available everywhere “by default”.

Managing software projects with git

- Git is a version control system. It is used to manage software projects … though I use it to manage complex writing projects (such as books) as well.
  - Once you learn git you'll wonder how you ever got by without it. It is that powerful
- You learn git by using it. To get us started, we will stick to the most basic usage.
- A repository is a holds text, code, and other resources the comprise a project. They are stored as "diffs" from a core master. This lets you move to past versions of the system should something break.
- You can work with git on your local system, but most of us use it as a distributed system hosted by a service such as gitub.
- You clone a copy of the repository to create a personal copy of the repository. For example, for our course:

  % git clone https://github.com/tgmattso/SciCompHPC.git
- To fetch an update to a cloned repository, you just type:

  % git pull
- If you own the repository or have right permission to it you update files with the commands:
  - git add list_of_one_or_more_files
  - git commit –m"message explaining what you are adding or updating"
  - git push

Git is the foundational tool professional software engineering. It can manage updates from multiple developers (pull requests), run automated testing workflows to validate commits and much more.

git clone https://github.com/tgmattso/SciCompHPC.git

Supporting our transition to hands-on learning

- Local systems (including your laptop) and how we will use them
- The Linux Operating System (OS) for HPC programmers
- Basic tools for HPC software developers working with Linux
- C … the high-level assembly code of computing. The simple, minimal subset.

What is C?

- C created in the 1970's at Bell Labs by Dennis Ritchie.
  - He was interested in low level, system programming and eventually implemented the Unix OS kernel in C.
- It is a small language. The keywords and concepts in C are "easy" to learn.
- It is a low level language that maps directly onto hardware.
  - It can be used in complex ways as a portable assembly language
- For example, what does this program do?

```
#include <stdio.h>
int main()
{
    union {
        char  what[16];
        int   cipher[4];
    } mystery,  *p;


    p = &mystery;
    p -> cipher[0] = 0x6c6c6568;
    p -> cipher[1] = 0x77202c6f;
    p -> cipher[2] = 0x646c726f;
    p -> cipher[3] = 0x0000000a;
    printf("%s",p->what);
}
```

A deliberately obnoxious program in C.

Source: "A Book on C", Al Kelley, Ira Pohl, 1984

% vi hello.c	**Use vi to type code into the file hello.c**
% gcc hello.c	**Compile hello.c to create executable, a.out**
% ./a.out	**Execute a.out located in current directory**
`hello, world`	**Output from the program**

What is C?

- C is an imperative language … you tell the computer what to do.

- C is a compiled language … the code goes in a file ending in **.c** and is compiled into an executable.

% vi hello.c	**Use vi to type code into the file hello.c**
% gcc hello.c	**Compile hello.c to create a.out**

Include a file that defines the contents of the standard input output library

```
#include <stdio.h>
int main()
{

   printf("Hello World\n");

   return 0;
}
```

main function with a single block

- The program is a collection of statements: … strings of characters that end with a semicolon.
- and preprocessor directives (starting with a #) … commands to the compiler resolved before compilation (such as “include” directives)

- C is a block structured language … code is organized into blocks between curly braces …. **{** and **}**.
  - A single statement is a block. The braces are optional in that case

Every program has a main function.

A function has a:
- Name … for example, **main**
- Return type … for example, **int**
- Zero or more arguments between parenthesis **()**
- A return statement … for example **return 0;**

What is C?

- In C you must declare variables before you use them … unlike python, C does not infer types
- We have the normal scaler types (int, long, float, double, char).
- We can define arrays with square backets around the size for each dimension and then index the elements of the array directly.
- A cast statement converts between types … it's the type you are converting into in parenthesis before the variable being converted (example: **(double)i**;

```
#include <stdio.h>
#define N 5
int main()
{
    int j, k=0;
    float r, f;
    double big=0.0, A[N];


    for(int i=0;i<N;i++){
        A[i]  = (double)i;
        big += A[i];
    }


    printf(" sum of A = %f\n",big);
    return 0;
}
```

It is bad form to have literals (such as the number 5) in your code. Give them a name and define them with a macro up front

What is C?   C loves pointers

- A variable is a name associated with a location in memory. The type and size associated with a variable defines how to interprets the contiguous block of bytes identified by the variable.
- We usually use pointers and memory managers to deal with arrays since in practice the size of the array varies from one execution to the next (often based on input data)
- This code is equivalent to that on the prior slide

```
#include <stdio.h>
#include <stdlib.h>
#define N 5
int main()
{
    int j, k=0;
    float r, f;
    double big=0.0, *A;

    A = (double*)malloc(N*sizeof(double));

    for(int i=0;i<N;i++){
        *(A+i) = (double)i;
        big += *(A+i);
    }

    printf(" sum of A = %f\n",big);
    return 0;
}
```

What is C?   Here are a bunch of details

- There are a collection of commonly used libraries in C. You tell the compiler/linker how to use them by including their header files:
  - #include<stdio.h>: The standard input/output library
  - #include<stdlib.h>: The standard library … mostly used for the malloc memory manager
  - #include<math.h>: A number of match functions including sqrt() and pow(). May require –lm for linker

- C supports the standard set of arithmetic operators you'd expect *, + , -, %, and /
  - % is the modulus operator … it returns the remainder of integer division: 17%5 = 2
  - / of floats and doubles returns what you'd expect. For integers, it truncates to an integer: 17/5 = 3
  - There is no exponentiation. Use the pow() function from math.h

- The formatted print function, printf(), takes a format sting and a list of variable. printf(" B = %d \n",B) where the backslash n (\n) says to start a new line. Common formats
  - %d: integers.   You can use %ld for long integers (that is variables declared as long)
  - %f: floating point numbers, You can use %lf for double precision numbers (variables declared as double)
  - %s: Strings.   These are variables declared as an array of characters (char)

- Logical operators ==, !=, <, and > which you use in logical expressions (for example if, else, and else if).

What is C? Functions

- C programs are organized around functions. main() is a function
- A function has a return type and arguments that are passed by value.
- The compiler/linker must know function interfaces at compile-time.
- Therefore:
  - Define functions before main (as with times2())
  - Define a function prototype and the function elsewhere (such as the end of the file … as with isOdd())

```
#include<stdio.h>
int isOdd (int);
int times2 (int D){ return 2*D;}
int main()
{
    int j=7, z;
    z = j;
    if (isOdd(j))
      z = times2(j);

    printf("An even number %d \n",z);
}
int isOdd(int j)
{
   if(2*(j/2)==j)
      return 0; // nonzero is true
   else
      return 1; // zero is false
}
```

```
% ./a.out
An even number 14
```

What is C? Functions

- This is the same program as before, but split between three files.

- Compile as follows where I assume all three files are in the same directory

  gcc –o DumbProg myprog,c myfuncs.c

- This is important when the functions in myfuncs.c are ones you want to reuse between many programs.

In file myprog.c

```
#include<stdio.h>
#include "myfuncs.h"
int main()
{
    int j=7, z;
    z = j;
    if (isOdd(j))
      z = times2(j);

    printf("An even number %d \n",z);
}
```

In file myfuncs.c

```
int times2 (int D){ return 2*D;}
int isOdd(int j)
{
   if(2*(j/2)==j)
      return 0; // nonzero is true
   else
      return 1; // zero is false
}
```

In file myfuncs.h

```
int times2 (int);
int isOdd(int);
```

What is C? Functions

- This is the same program as before, but now we pass the result of the times2() function through an argument.
- Functions are “pass by value”. So if you want an input value to change, you pass in the address of the variable with the address-of operator, &
- Then change the function to work with a pointer to memory … D is the pointer, *D is the value pointed to by D.

In file myprog.c

```
#include<stdio.h>
#include "myfuncs.h"
int main()
{
    int j=7, z;
    z = j;
    if (isOdd(j))
      times2(&j);

    printf("An even number %d \n",z);
}
```

In file myfuncs.c

```
int times2 (int *D){*D= *D * 2; return 0;}
int isOdd(int j)
{
   if(2*(j/2)==j)
      return 0; // nonzero is true
   else
      return 1; // zero is false
}
```

In file myfuncs.h

```
int times2 (int*);
int isOdd(int);
```

What is C? Scope

- A variable is a name for a location in memory.
- The scope of a variable is the region of code where that variable is visible (that is, able to be read or written).
- That region is the basic bock where a block is the curly braces.
- A variable defined inside a block “masks” any variables of the same name declared outside the block. When you leave that block, variables defined in the block “go away”
- Filescope variables declared outside any functions are visible to all the functions in the file (that is, in that compilation unit).

```
#include<stdio.h>
int z = 5;
int main()
{
    int j=7;

    printf(" j and z before = %d %d\n",j,z);
    {
       int j=8;
       z = z+2;
       printf(" j and z inside = %d %d\n",j,z);
    }
    printf(" j and z after = %d %d\n",j,z);

}
```

```
% ./a.out
 j and z before = 7 5
 j and z inside = 8 7
 j and z after = 7 7
```

One last thing before we move to writing parallel code.   This advice comes from a leader in software engineering practices for HPC.

Ignore his advice at your peril!!!

Best Practices for Scientific Computing

1. Write programs for people, not computers.
   - A program should not require its readers to hold more than a handful of facts in memory at once.
   - Make names consistent, distinctive, and meaningful.
   - Make code style and formatting consistent.
2. Let the computer do the work.
   - Make the computer repeat tasks.
   - Save recent commands in a file for re-use.
   - Use a build tool to automate workflows.
3. Make incremental changes.
   - Work in small steps with frequent feedback and course correction.
   - Use a version control system.
   - Put everything that has been created manually in version control.
4. Don't repeat yourself (or others).
   - Every piece of data must have a single. authoritative representation in the system.
   - Modularize code rather than copying and pasting.
   - Re-use code instead of rewriting it.
5. Plan for mistakes.
   - Add assertions to programs to check their operation.
   - Use an off-the-shelf unit testing library.
   - Turn bugs into test cases.
   - Use a symbolic debugger.
6. Optimize software only after it works correctly.
   - Use a profiler to identify bottlenecks.
   - Write code in the highest-level language possible.
7. Document design and purpose, not mechanics.
   - Document interfaces and reasons, not implementations.
   - Refactor code in preference to explaining how it works.
   - Embed the documentation for a piece of software in that software.
8. Collaborate.
   - Use pre-merge code reviews.
   - Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
   - Use an issue tracking tool.

Wilson, et al., (2014) https://doi.org/10.1371/journal.pbio.1001745 18

Source: David Bernholdt of ORNL, lecture at ATPESC’23

So now, at last, we are ready to learn how to write parallel code