02_Working_in_Linux

January 14, 2020

Module 02: Working in a Linux Environment: Cluster Computer Architecture

This module is largely focussed on building an understanding of the fundamental components of cluster computer architectures and establishing a common computing framework for working on Mines computer clusters.

Many scientific/geophysical problems require computing power and storage that is beyond what is available on a single computer. In such cases, scientists often parallelize their algorithms (i.e., divide the task into smaller chunks) and run smaller jobs on multiple computers simultaneously. This module introduces the computing resources available for students at Mines and shows how students can access and utilize these resources to speed up their large-scale jobs.

0.1 What is Linux

Linux is the most popular operating system for a growing number of scientific applications, especially in geophysics. Therefore, familiarity with Linux will be advantageous throughout your career as a geophysicist, especially when dealing with large-scale numerical problems that require high-performance computing solutions.

Linux offers a free operating system. Being open-source, anybody with programming knowledge can modify it.

In addition to being a free operating system, Linux is also a great platform for programming and supports almost all programming languages people use in geophysical computing. It offers free powerful tools ranging from simple text editors to powerful integrated development environments (IDE).

title

0.1.1 Basic Linux commands

Even though many Linux distros come with a GUI, we usually interact with Linux computers through a terminal. Here, we will remember/learn some of the terminal commands for the most frequent tasks we perform on our computers: creating, copying, moving, and deleting files and folders.

0.1.2 Changing directories

If you want to change your current directory, use the **'cd'** command.

The top level directory on any Linux system is the root directory. It contains all other directories and is designated by a forward slash **'/'**. Hence, every path in Linux starts with the **'/'** symbol (e.g., /home/tugrul).

```
cd /home/tugrul
```

To navigate up one directory, use **'cd ..'**
**
Figure 1. An example Linux directory structure.
**

0.1.3 Present Working Directory

It is always useful to know where you are. You can determine the directory you are currently browsing by typing the following command

```
pwd
```

0.1.4 Listing files in a directory

To see the list of files in your current directory, use the **'ls'** command.

```
ls -l
```

0.1.5 Copying files

you can use the **"cp"** command to copy files on a Linux system. For example, the command below copies the **"main.c"** file in the **"/home/tugrul/"** directory to **"/home/jeff/"** directory and saves with the name **"main_copy.c"**.

```
cp /home/tugrul/main.c home/jeff/main_copy.c
```

0.1.6 Creating directories

On a Linux operating system, directories can be created with the **"mkdir directory"** command.

0.1.7 Deleting files and directories

Files and directories can be deleted using the **'rm'** command.

For example, to delete a file, you can type

```
rm filename
```

When deleting directories, you have two options. The first one is a variant of the **rm** command, but for directories

```
rmdir directory_you_want_to_delete
```

Alternatively, you can provide the **"recursive"** option to the **rm** command to delete a directory and everything inside it:

```
rm -r directory_you_want_to_delete
```

0.1.8 Moving/Renaming files and directories

Files and directories can be moved and renamed using the **'mv'** command.

To rename a directory or file, try

```
mv old_filename new_filename
```

0.1.9 Creating and using aliases

A shell alias is a shortcut to reference a command. Instead of writing long commands every time, you can create short aliases that are shorter and easier to write. The standard syntax for creating alias is given below:

```
alias shortname="your custom command here"
```

A good use of aliases for your case is when accesing the Mio cluster. Instead of typing

```
ssh -Y username@mio.mines.edu
```

each time you want to connect to Mio, you can create an alias such that

```
alias mio="ssh -Y username@mio.mines.edu"
```

In order to make aliases permanent, you need to put that into your ~/.bash_profile or ~/.bashrc file.

Another good use of aliases is to avoid typing error when using Madagascar. You can alias the common spelling mistakes for the **scons** command to ensure that it will work most of the time without requiring a correction. An example is given below:

```
# scons aliases
alias scosn="scons"
alias scson="scons"
alias csosn="scons"
alias scns="scons"
alias sconsc="scons -c"
alias sconscv="scons -c; scons view"
alias sconscs="scons -c; scons"
alias sconscv="scons -c; scons view"
```

0.1.10 top

On Linux systems, the **top** command shows the Linux processes. It provides a dynamic real-time view of the running system.

As soon as you will run this command, you will see an interactive command mode where the top half oprtion will contains the statistics of processes and the system resource usage. And the lower half shows a list of the currently running processes. To exit from this window, you just need to press the *q* button.

0.1.11 Piping in linux

A pipe is a form of redirection used on Linux-based systems to send the output of one command, program, or process to to another command, program, or process as input for further processing.You can connect programs by piping using the pipe character **"|"**. The syntax is as follows

```
command_1 | command_2 | command_3 | ... | command_N
```

0.1.12 grep

The grep command searches file for a particular pattern of characters, and displays all lines that contain that pattern. For instance, in a directory with so many files, you can list all the files that contain the word "apple" by just typing the following command

```
ls -l | grep apple
```

Here, the **ls -l** command lists all the files in the current directory and sends all the listed file names to the **grep** command through pipe. The **grep** program then lists only the file names that contain the word "apple".

The **grep** command is by default case sensitive. Howeverm you can force it to make a case-insensitive search by adding the **-i** option.

0.1.13 find

You can use the **find** command find files on your Linux system. If used with the right arguments/options, find command is very powerful. The standard syntax is as follows

```
find directory_to_search -name "file_name_to_be_found"
```

You are not limited to search for files only using their names. In addition, you can also search for directories. Here are some examples on how you can search files on Linux platforms

Find all file in the entire system that are names "test.txt":

```
find / -name "test.txt"
```

Find all directories in the system called "log":

```
find / -type d -name "log"
```

Find all files in the "/usr/bin" directory of subdirectories that are larger than 1 megabyte:

```
find /usr/bin -size 1M
```

Find all files in the "/usr/bin" directory that are created more than a day ago:

```
find /usr/bin -mtime 1
```

Find all files in the "/usr/bin" directory that are created less than a day ago:

```
find /usr/bin -mtime -1
```

Find all files in the "/usr/bin directory" that are owned by the user "esteban"

```
find /usr/bin -user esteban
```

tar The Linux 'tar' stands for tape archive, is used to create Archive and extract the Archive files. tar command in Linux is one of the important command which provides archiving functionality in Linux. We can use Linux tar command to create compressed or uncompressed Archive files and also maintain and modify them.

To create a tar archive, you can write

```
tar -zcvf mytarfile.tar /user/myname
tar -zcvf mytarfile.tar /user/myname "fileA" "fileB"
```

To extract a tar archive, just type

```
tar -zxvf mytarfile.tar extracted_tar_archive
```

0.2 What is a cluster?

Frequently, real-life three-dimensional geophysical problems are much larger than a single laptop or computer can handle. In such cases, we inherently need to utilize more computers to have enough computing power and memory to be able to solve the problem at hand within a reasonable time frame. Luckily, in many universities, companies, and national labs, there are large-scale computing systems that are really just collection of computers that work together to solve problems larger than any one computer can solve. These cluster of computers are often interchangably called **"clusters"**, **"supercomputers"** or resources for **"high-performance computing (HPC)"**.

The individual computers in a **cluster** are often referred to as **"nodes"**. And, just as we, people, need to talk to each other in order to collaborate, nodes in a cluster also need to be able to talk to one another; hence nodes in a cluster are connected together with a computer network (often referred to as interconnect).

Just like your laptop or desktop computers, the fundamental components of each node in a cluster are nothing but processors, memory, disks, and so on. The primary difference is the scale: nodes in clusters usually have faster processors, more memory, and more storage. And, just like your personal computer, each nodes needs an operating system to be able to function properly.

Today, computer clusters are widely used by many institutions, companies and labs. Top500,org publishes a list of the most powerful 500 supercomputers in the world along with interesting statistics twice a year. Figure 2 shows the most recent list to date that is published in November 2019.

Figure 2. The top 500 most powerful clusters in the world as of November, 2019 .

Figure 3. HPC usage by countries in terms of volume (left) and the processing power(right) as of November 2019 (TOP500).

Linux is currently the most popular operating system for clusters. Figure 4 shows the operating systems that are installed on the 500 most powerful computer systems in the world. Another good reason to learn Linux!

Figure 4. Operating systems used by the 500 most powerful computer systems in the world as of November 2019 (TOP500).

Although high-performance computers have faster processors, more memory and storage, the real power comes from using the resources in parallel than in serial. Then, the main challenge of a computational geophysicist is to rearrange the algorithms so that the large-scale problem can be divided into smaller chunks and each node can work on a smaller portion of the problem. Having a deeper understanding of the geophysical algorithms will be greatly helpful in these scenarios.

0.2.1 High-Performance Computing at Mines

Mines has a number of high-performance computing platforms: Wendian, Mio, Mc2, AuN. You can find more info on the specifications at https://hpc.mines.edu/ .

In this class, we will use Mio (see Figure 2), which has approximately 120-plus Tflop computing power. You can see the hardware on Mio below:

- 178 compute nodes
- 8-28 compute cores per node
- 2.4GHz – 3.06GHz
- 24-256 GB/Node
- Infiniband Interconnect
- 2 GPU nodes – 7.23 Tflops
- 240 TB parallel file system

Figure 3. The CSM Mio cluster.

0.3 The anatomy of a Computer Cluster

Computer clusters often include several different types of nodes, which are used for different purposes. **Master** or **Head** nodes are where you login to interact with the HPC system. master node is also the place to download/retrieve data from HPC to your local computers.

Compute nodes are where the actual heavy computing takes place. In a typical cluster setup, user usually cannot access compute nodes directly as access to these nodes are generally controlled by a scheduler.

**

Figure 4. A high-level overview of a computer cluster.

**

0.3.1 Nodes

Each node in a cluster is an individual computer with its processor and memory. Modern processors contain multiple cores that are responsible for the actual computations. In addition, there are various fast memory caches for holding the data that is currently being worked on. On the CSM Mio system, nodes have 8 or 28 cores. Often, nodes in cluster have multiple processors, which effectively doubles the number of cores per node.

Each node also has a certain amount of random access memory (RAM) available. on Mio, each compute node has 24 or 256 GB of memory.

In addition, each node has access to a disk (also called "storage" or "file system") that is usually shared among all nodes

**

Figure 4. A typical node scheme.

**

0.3.2 Storage

Usually problems people work on clusters are huge and involves many large files. In addition, these files have to be accessible from all nodes available on the system. Hence, most cluster have specialized file systems that are designed to meet these needs. Often, these file systems are intended to be used only for short- or medium-term storage. As a consequence, HPC systems may have several different file systems available, each serving for a different purpose (e.g., home and scratch file systems).

DO NOT PUT THE ONLY COPY OF YOUR SOURCE CODE TO SCRATCH!

0.3.3 The Scheduler

Computer clusters are usually shared by many users and it is common to allocate subsets of resources to different tasks. To manage the sharing of available resources amoing all of the jobs, HPC systems usually use a scheduler. Users generally, login to the master node and submit their jobs through a scheduler which runs the computation on the compute nodes. Often, schedulers decide which job to run first based on their priority that is usually assigned by users or the administrator.

0.4 Memory

The individual CPU-cores can be put together to form large parallel machines in two fundemantally different ways: the shared and distributed memory architectures.

0.4.1 Shared Memory

There are two different ways to consider regarding the terms shared memory and distributed memory: in terms of how the hardware actually implemented, and what do they mean in as programming abstractions.

From a strictly hardware point of view, we see a typical shared-memory architecture where all processors can access to a common physical memory. If you have multiple processors, then those would be able to access exactly the same memory locations (see Figure 5). This architecture is very similar to what you have in your personal computers. Hence, developing high-performance parallel algorithms for shared-memory systems is straightforward and very similar to developing serial applications that you would normally run on your laptop. However, there are two big problems with this approach: 1) building CPUs with hundred thousands of cores is not possible for now; 2) all CPU-cores compete for access to memory over a shared bus. That's this kind of systems are not manufactured today.

However, the idea of communicating directly through memory remains a useful programming abstraction. So in many systems, programmers use common programming techniques (e.g., OpenMP) to perform parallel tasks can directly access the same **"virtual"** memory regardless of where the physical memory actually exists. So, this programming model can be thought as an illusion that all memory is actually shared.

**

Figure 5. Shared memory architecture.

**

0.4.2 Distributed Memory

From hardwave point of view, in distributed memory systems, processors perform computations on their local memory and use networks to transfer data with remote processors. Although this model greatly simplifies the hardware implementation, the programming for these platforms becomes more complex as programmers also has to take of the communication between the processors in addition to actual computations.

As in the shared memory case, distributed memory can also be used as a programming model and CPU-cores can be forced to communicate through the network and behave like they are accessing their local memory even though they actually see the same common memory. So, you can use your CPU cores on your laptop as if they are on different machines.

**

Figure 6. Distributed memory architecture.

**

0.5 Connecting to the Mio System

0.5.1 Habits of a responsible HPC citizen

Before, accessing the Mio system, just bare in mind that Mio is being shared by many researchers and if you follow some basics rules, you may avoid receiving angry emails from the admin and other students or researchers.

When using the Mio system, remember the habits of a responsible HPC user.

A responsible user:

- does not use the head node for heavy calculations
- is careful about the resource usage (nodes and disk space)
- knows what to keep/delete and where to keep
- is respectful of other people's data.

0.5.2 Accessing Mio

Now that we are familiar with the high-performance computing systems and know what to expect after we access them, it is a good time to put what we learned in the previous section into practice.

Connecting to clusters is often done through a terminal using a tool known as **"SSH"** Hence, the first step of accessing Mio is opening a terminal (assuming you already have a Mio account. Otherwise, you can get an account by emailing hpcinfo@mines.edu).

Assuming you are on campus, or connected to Mines network via VPN, you can get to mo by entering the following in a terminal window:

```
ssh -Y username@mio.mines.edu
```

You will be asked for your password. The password required here is your MultiPass password.

After entering your password, you should be logged in to the **head** node and see a prompt like the one below.

[yourusername@mio001 ~]$

Since, Mio uses a Linux operating system, you can use all the commands you learned in the Linux section.

0.6 Setting up your environment

Because HPC systems serve many users with different software needs, HPC systems often have multiple versions of commonly used software packages installed. Since you cannot easily install and use different versions of a package at the same time without causing potential issues, HPC systems often use environment modules (often shortened to modules) that allow you to configure your software environment with the particular versions of software that you need.

On Mio, you see a list of all the available modules using the command:

```
module avail
```

Many HPC systems also have a custom environment that means that binary software package will not simply work "out-of-the-box". They may need differnt options or settings in your job script to make them work. We will see what kinds of settings we need to make in order to be able to use Magagascar on Mio.

0.7 Transferring files between local computer and Mio

To copy a file to or from Mio, you can use the **'scp'** command.

To transfer a file from your local computer to Mio:

```
scp /path/to/local/file.txt username@mio.mines.edu:/path/on/Mio
```

To download/copy a file from Mio to your local computer:

```
scp username@mio.mines.edu:/path/on/Mio/file.txt  /path/to/local/directory
```

0.8 Best advice to the programmers

Version control systems are commonly used by professional programmers. So many times, you will change something in your program and everything will stop running. Or, you may accidentally delete your source code and may have to face catastrophic consequences. To avoid all these problems, it is crucial to be aware of the availability of the version control systems and to use them regularly.

Git is the msot commonly used version control system, It tracks the changes you make to files, so you have a record of what has been done, and you can revert to specific versions should you ever need to. You have different platforms that provide such services (e.d., github, bitbucker, gitlab).

0.9 Using Madagascar on Mio

Working on the CSM Mio cluster requires developing new skills (for some!) about working in a Linux environment; writing and compiling your code in a lower-level language (e.g., C/C++/F90); submitting jobs through a job scheduler (e.g., SLURM); and debugging, validating and optimizing your code to obtain the best results in the least amount of time.

To help facilitate this process, we are going to perform our tasks within the Madagascar framework. While not a perfect environment for doing everything you might want to do in a computational sense, it does offer an extensive API that allows one to interact with (i.e., read in and write out) multi-dimensional array data stored in a regularly sampled format (RSF). There are also lots of code examples that provide a good starting point for developing your own applications.

Fortunately, there is a shared workspace in Mio, which you can access and use Madagascar by just seeting some environment variables.

To run Madagascar and have all the correct environment variables, you will need to set up your bash environment. There is a script prepared for you under **/gpfs/lb/sets/geop/M8R/"** directory.

If you have never used Mio before, you can set up your environment by typing the following commands in order.

-

```
mv ~/.bash_profile ~/bash_profile_old
```

-

```
cp /gpfs/lb/sets/geop/M8R/bashrc_generic_geop ~/.bash_profile
```

-

```
source ~/.bash_profile
```

If you are an existing user, and already have a bash setup, you can just copy and paster the contents of the **/gpfs/lb/sets/geop/M8R/bashrc_generic_geop** file to your **bash_profile** file. The most important lines

```
alias scons=/opt/python/gcc/2.7.11/bin/scons

## . . Load Mio utilities
module load utility >& /dev/null

##### Madagascar
# . . PYTHON
module load PrgEnv/python/gcc/2.7.11
# . . MPI
module load openmpi/gcc

source /gpfs/lb/sets/geop/M8R/RSF2.0/RSFSRC/env.sh

## . . Update PYTHONPATH
export PYTHONPATH=$RSFSRC/book/Recipes:/gpfs/lb/sets/geop/M8R/lib/scons-2.5.1:${RSFROOT}/lib/pyt

## . . Update PATH
export PATH=${RSFSRC}/book/Recipes/:/gpfs/lb/sets/geop/M8R/lib/scons-2.5.1/SCons:${PATH}

## . . RSF manual pages
export MANPATH=/gpfs/lb/sets/geop/M8R/RSF2.0/share/man

## . . Need to define this path
export GEOP=/gpfs/lb/sets/geop/
```

0.10 A really brief introduction to C

C is a general-purpose programming language that was mainly developed as a system programming language to write an operating system. The main features of C language include low-level access to memory, a simple set of keywords, and clean style. These features make C language suitable for system programmings like an operating system or compiler development as well as scientific programming.

The C language program can be modeled as a state machine. The state is defined by the values assigned to the variables declared in the program. The program code specifies how its state should change during program execution, which can be made by four fundamental primitives: assignment, sequential composition, branching, and loop.

Many programming languages have borrowed syntax/features directly or indirectly from C language. Like syntax of Java, PHP, JavaScript, and many other languages are mainly based on C language.

0.10.1 The structure of a C program

Any program in C can be written in the structure given below. Often, you will need to use more advanced programming concepts such as functions to improve the readability and repeatability of your code. We will see and use some of these concepts in examples below.

The structure of a typical C program is as follows:

**

Figure 7. Structure of a C program.

**

The components of the above structure are explained below.

Including header files In a C program, the first thing you do is to include the Header files. A header file is a file with extension **".h"**, and contains C function declarations and definitions. Some of the useful header files are

- stdio.h - contains core input and output functions
- stdlib.h - provides functions for mathematical computations, I/O, memory management, and several other OS services.
- math.h - defines common mathematical functions
- string.h - containt functions for string manipulation operations
- time.h - contains functions related to date and time

The syntax to include a header file in C:

```
#include <headerfile.h>
```

Main method declaration Every C program has to have a main function. This is the starting point of your program.

The syntax to declare the main function is:

```
int main()
{}
```

Variable declaration Usually, the first part of your main function is variable declaration. Here, you define the names and types of the variables that you will use later in your program.

Below, you can find an example of defining a floating point variable (declared using the **float** keyword in C) named "apple".

```
int main()
{
    float apple;
}
```

Code (Body) of the program Body of a C program refers to the operations that are performed in the program. It can be anything like manipulations, mathematical operations, or printing the value of a variable.

Below, you can see an example where the program writes the value of an integer variable named "dontdothis"

```
int main()
{
    int dontdothis;

    dontdothis = 8;

    printf("%d", dontdothis);
}
```

The return statement The last part of any C program is the return statement. In C language, the **return** statement refers to the the returning values from a function. the return statement and the return value depend on the return type of the function. For example, the function returns an integer number, which is by the ***int*** keyword.

Example:

```
int main()
{
    float a;

    printf("%f",a);

    return 0;
}
```

In cases when the return type of a C function is **void**, there will be no return statement.

Comments Interestingly, computer programs are also meant to be read by other people. To help others understand the code one should describe using plain language what it does and any assumptions the code relies on. Such descriptions are are called **comments** in computer programming and can be written in two ways in C language.

For a single line comment, you can use

```
// this is a comment
```

Sometimes, you will need write long explanations in your code. In such cases, you may want to use more than one line. You can add multiline comments in C as shown below:

```
/* This is short a comment */

/*
But,
can
be
a
long
one
if
you
want
!
*/
```

0.10.2 Writing your first C program

Since we know the main components of a C program, we can go ahead follow the biggest cliche in computer programming: writing the infamous "Hello, World" program.

You can find the source code below:

```
/*
 * First C program that says Hello (hello.c)
 */
#include <stdio.h>  // Needed to perform IO operations

int main()
{                       // Program entry point
    printf("Hello, world!\n");  // Says Hello
    return 0;                   // Terminate main()
}
```

C is a compiled language. So, your source code needs to be converted to a form that the computer hardware can understand. We use compilers for this purpose. "gcc" is the standard compiler for C that comes with Linux. To compiler your C code with gcc, you can use the gcc compiler as shown below

```
gcc -o binary_name source_file.c
```

To compile and execute the Hello, Word program, just write the code above to a file named **"hello.c"** and compile your program as shown below

```
gcc -o hello hello.c
```

To execute the program, simply type

```
./hello
```

0.10.3 Arrays in C

Hello, World program is exciting! However, you often will want to write something more useful such as finding out the average of 100 integer numbers. In this case, you have two ways to do this: 1) Define 100 variables with int data type to store the values in the variables and then calculate the average of them. 2) Have a single integer array to store all the values, loop the array to calculate the average. Obviously as in the second solution, it is more convenient to store same data types in one single variable and later access them using array index.

An array in the C language represents **fixed** size sequences of values of the same type. An array variable is defined by the type of the values it contains and its size.

For example, the code below declares an array variable called "point" which contains 3 integer values.

```
float point[3];
```

You can access the elements of an array in C using the array index between square brackets. For example:

```
float mydata[20];

mydata[0] /* first element of the mydata array */
mydata[19] /* last (20th) element of the mydata array */
```

You can also create multidimensional arrays in C. For example

```
float x[3][4];
```

Here, **x** is a two-dimensional (2D) array with 3 rows and each row has 4 columns.
You can initialize a 2D array in different ways:

```
/* each {} is a row */
int c[2][3] = {{1, 3, 0},{-1, 5, 9}};

/* 2 rows and 3 columns= we have a total of 6 elements */
int c[2][3] = {1, 3, 0,-1, 5, 9};
```

0.10.4 for loop in C programming

This is the most frequently used loop in C programming. The syntax of for loop is as follows:

```
for (initialization; condition test; increment or decrement)
{
    // statements to be executed repeatedly
}
```

Below, you can see an example C program that writes integer number from 3 to 7:

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

```
        int i;
        for (i=3; i<=7; i++) /* in C, i++ is equal to i=i+1 */
        {
            printf ("%d \n", i);
        }
        return 0;
    }
```

Especially when solving multidimensional problems, you will often need nested for loops. You can find an example of a nested for loop below

```
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 2; j++)
    {
        /* the values of i and j will be as follows
         * i = 0, j = 0
         * i = 0, j = 1
         * i = 1, j = 0
         * i = 1, j = 1
        */
    }
}
```

Conditional branching in C (if … if else) Sometimes, you will need to execute a block of code only when a given condition is true or false. In such cases, you can use the **"if"** statements as shown below:

```
    if (condition)
    {
     // block of C statements here
     // these statements will only be executed if the condition is true
    }
```

You can write more complicated examples by adding more conditions using the **"else"'** and **"else if"** statemets as shown below:

```
    if (i > 40) {
        /* execute the code written here */
        x = i;
        y = 2*i;
    } else if (i < 40) {
        /* only execute the code written here */
        x = -i;
        y = 3*i;
    } else {
        /* forget the code above, set them all to zero */
```

```
        x = 0;
        y = 0;
    }
```

Example C program to find out the average of 4 integers

```
#include <stdio.h>
int main()
{
    int avg = 0;
    int sum = 0;
    int i=0;

    /* declare an integer array 4 elements */
    int arr[4] = {3, 8, 24, 16};

    /*
     * We are using a for loop to sum all the values in the array
     * and store this sum in the "sum" variable
     */
    for (i=0; i<4;i++)
    {
        sum = sum + arr[i];
    }

    /* duh */
    avg = sum / 4;
    printf("Average of 4 numbers is: %d \n", avg);
    return 0;
}
```

Again, you can compile the above program as

```
gcc -o avg4vec avg4vec.c
```

And, execute

```
./avg4vec
```

Q: Did you realize the potential bug in the above program?

Example program for the sum of two matrices The program below finds the sum of two 2x2 matrices

```
/* C program to find the sum of two matrices of order 2*2 */

#include <stdio.h>
```

```c
int main()
{

  /* declare arrays */
  float a[2][2], b[2][2], result[2][2];

  /* assign the values of arrays */
  /* you can also define the loop counter variable (i) here */

  for (int i = 0; i < 2; i++)
  {
    for (int j = 0; j < 2; j++)
    {
      /* a matrix */
      a[i][j] = i*j;

      /* b matrix */
      b[i][j] = i + j;

    }
  }

  /* adding corresponding elements of two arrays */
  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 2; ++j)
    {
      result[i][j] = a[i][j] + b[i][j];
    }
  }

  /* display the sum */
  printf("\nSum Of Matrices: \n");
  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 2; ++j)
    {
      printf("%f ", result[i][j]);
      if (j == 1)
      {
        printf("\n");
      }
    }
  }

  return 0;
}
```

0.10.5 More advanced concepts in C

Pointers Arrays are quite useful, but as you might have realized, we need to know their size before compiling our programs so that the compiler can allocate the memory required to store those arrays. However, in real-life situations, sizes of arrays you will use in your programs will vary depending on the size of the data you are processing. C language provides some functionality to dynamically allocate arrays whose size can be decided during the runtime. Before, we will be able to utilize that functionality, however, we need to understand a very famous programming concept: pointers.

So far in your programs, you assigned values to your variables, and performed some mathematical operations to change/use those values.

For example:

```
int a = 5;
int b;

b = a*a;
```

However, in C, you can access other properties of variables such the address of the memory location that variable is being stored. To do that, we use the **"&"** operator before the variable name.

Below is an example program that writes both the value and the address of an integer variable:

```
#include <stdio.h>
int main()
{
   int num = 10;
   printf("Value of variable num is: %d \n", num);
   /* To print the address of a variable we use %p
    * format specifier and ampersand (&) sign just
    * before the variable name like &num.
    */
   printf("Address of variable num is: %p \n", &num);
   return 0;
}
```

In the program above, we have a variable num of int type. The value of num is 10 and this value must be stored somewhere in the memory. Whenever you define a variable, a memory space is allocated to store the value of that variable. And, same as our house has an address, these memory locations also have their addresses. The **"Address of (&)"** symbol simply shows us that address. Then, we can ask an important question: can we define a variable that keeps the address of another variable? The answer is yes, and we call such variables as **pointers** in C programming language.

In C language, we declare pointers using the **"Value at Address (*)"** symbol. The code below shows an example on how pointers are declared and used.

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

```
    //Variable declaration
    int num = 10;

    //Pointer declaration
    int *p;

    //Assigning address of num to the pointer p
    p = &num;

    printf("Address of variable num is: %p", p);
    return 0;
}
```

In the program above, we first declare an integer pointer variable “p”, then assign the memory address of the integer “num” variable to this pointer. That’s why when we print the value of “p”, the program shows us the address. To access the actual value at the memory address that the pointer variable points to, you can add "*" symbol before the pointer variable.

For example

```
printf("Address of variable num is: %p", p);
printf("Value of variable num is: %d", *p);
```

An important point to note here is that the data type of pointer and variable must matchm an int pointer can hold the address of an it variable, similarly, a pointer declared with float data type can hold the address of a float variable.

Below, you can find another example demonstrating the usage of the **“Address of (&)”** and the **"Value at Address (*)"** operators:

```
#include <stdio.h>
int main()
{
    /* Pointer of integer type, this can hold the
     * address of a integer type variable.
     */
    int *p;

    int var = 10;

    /* Assigning the address of variable var to the pointer
     * p. The p can hold the address of var because var is
     * an integer type variable.
     */
    p= &var;

    printf("Value of variable var is: %d", var);
    printf("\nValue of variable var is: %d", *p);
    printf("\nAddress of variable var is: %p", &var);
    printf("\nAddress of variable var is: %p", p);
    printf("\nAddress of pointer p is: %p", &p);
```

```
   return 0;
}
```

Pointers and arrays There are many things you can do with pointers. Similar to the single variables, pointers also work with arrays. Below is an example demonstrates the usage of pointers with arrays.

```
#include <stdio.h>
int main( )
{
   int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;

    /* pointer declaration */
   int *p;

   /* assign the address of the first element to p pointer */
   p = &val[0];

   /* for loop to print value and address of each element of array*/
   for ( int i = 0 ; i < 7 ; i++ )
   {
       /* The correct way of displaying the address would be using %p format
       * specifier like this:
       * printf("val[%d]: value is %d and address is %p\n", i, val[i], &val[i]);
       * Just to demonstrate that the array elements are stored in contiguous
       * locations, I am displaying the addresses in integer
       */
      printf("val[%d]: value is %d and address is %d\n", i, val[i],(unsigned int) p);

      /* magic */
      p++;
   }
   return 0;
}
```

The example above shows how pointers can be used with arrays. An important observation here is that the difference between the memory address of each element in the array is 4. The reason for that is each element in the array is an integer value, which takes 4 bytes of memory to store on our system. Since C language keeps the elements of an array in consecutive contiguous memory locations in the memory, the difference between the memory address of each consecutive element is 4.

Another important observation one can make in the above example is how increasing the pointer value by one moves 4 bytes in the memory. Again, the reason for that is our array is defined of int type and each int variable takes 4-bytes in the computer memory. So naturally, each element in the array is four byte away from the previous one, and hence, memory address is increased by 4 when you increase the pointer by one.

Dynamic memory allocation in C Now, we are aware of the fact that we can access any array element once we know the memory address of the first element in the array and the type of the array, we can all this knowledge to simulate a dynamically allocated array in C.

The main idea here is to define a pointer to the the memory address of the first element of an array and tell the compiler to reserve us some memory space starting from that address. We will also tell the compiler how much memory space we need, so we will end up having enough memory space for us to store all our array in contiguous locations in the memory.

Although, this process sounds somewhat complicated, C programming language provides us some functions to achieve these tasks with minimal pain. You can even perform all these operations with just a single line of command!

In C language, the **"malloc"** or **"memory allocation"** methods is used to dynamically allocate a single large block of memory with the specified size. The **"malloc"** methid returns a pointer type of void which can be cast into o pointer of any form.

The standard syntax for the **malloc** command is as shown below

```
ptr = (cast-type*) malloc(byte-size)
```

For example:

```
int* ptr = (int*) malloc(100 * sizeof(int));
```

In the code above, since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer "ptr" hold the address of the first byte in the allocated memory. Here, instead of assuming the size of an int variable as 4 bytes, we use the **"sizeof"** function to get the size of an **int** type variable. This can be considered as a safer approach since the size of each type can vary accross different hardware and operating systems.

If the memory space is insufficient, allocation fails and malloc returns a NULL pointer.

And, since we tricked the compiler when allocating the memory dynamically, it is our reponsibility to clean our mess and clean this memory space. We do this using the **"free"** method at the end of our program.

For example

```
/* allocate 400 bytes of memory */
int* ptr = (int*) malloc(100 * sizeof(int));
/* clean/free/release that memory space */
free(ptr);
```

Below, you can see an example demonstrating the dynamic memory allocation in C. When you execute the program, it will ask you to specify the number of elements in the array. Then, the program will dynamically allocate this array, assign the values of each element, and dump them to your terminal.

```
#include <stdio.h>
#include <stdlib.h>  /* we need this for malloc */

int main()
{

    // This pointer will hold the
```

```
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    printf("Enter number of elements:\n");
    scanf("%d", &n); // opposite of printf, but takes/reads input from user/terminal

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
        /* skip one line */
        printf("\n");
    }

    return 0;
}
```

C File I/O When a C program is terminated, all the data generated by that program is lost. Often, you will need to store some information in a file for later use. In addition, if you have to enter a large number of data, it will take a lot of time to enter them all through terminal. However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C and move your data from one coputer to another without any changes.

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
FILE *fptr;
```

Once you declare a pointer, you can open a file for reading or writing modes using the **"fopen"** method.

The syntax for opening a file in C is:

```
fptr = fopen("/path/to/file/filename",mode");
```

Here the **"mode"** defines what we can do with the file defined as the first parameter to the **fopen** function. To open a file to read its contents, you can open the file in the **reading** mode by providing the **"r"** character as the second argument to the **fopen** function. Similarly, opening a file in the **"w"** mode allows users to write data into that file.

For example

```
FILE *fptr_in;
FILE *fptr_out;

/* opens the file in the read mode */
fptr_in  = fopen("readFromThisFile.txt,"r");

/* opens the file in the write mode */
fptr_out = fopen("writeToThisFile.txt,"w");
```

Writing and reading data from files are done by **"fprintf"** and **"fscanf"** methods, respectively.

For example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int num;
   FILE *fptr_in;
   FILE *fptr_out;

   /* use appropriate location if you are using MacOS or Linux */
   /* input file */
   fptr_in = fopen("input.txt","r");

   /* output file */
   fptr_out = fopen("output.txt","w");

   /* read an integer number from the input file and store in the num variable */
   fscanf(fptr_in, "%d", &num);

   /* we read our value, so we can close the input file here */
   /* close the output file */
   fclose(fptr_in);

   /* write the number to file */
   fprintf(fptr_out,"%d",num);
```

```
    /* close the output file */
    fclose(fptr_out);

    return 0;
}
```

In the program above, user provides a file named **"input.txt"** that contains an integer number in it. The program then opens this file, reads the value written in it, and writes that value to another file called **"output.txt"**. See how both files are closed calling the **"fclose"** method!

As you may have realized, the functions we use for read from a file or writing to a file are quite similar to the functions we have been using to read from or write to terminal. The only difference is that the methods associated with file operations start with **"f** and takes the file pointer as the first argument.

Improving the program that finds out the average value of a vector Since we know how to allocate dynamic arrays and using files to read and write data, we can modify the program we wrote earlier to calculate the average values of a vector. Here, we will make two modifications: 1) the program will read the length of the vector as well as the values of each element from a file provided by the user; 2) the calculated average value will be stored in a text file.

The input file, **"vec_in.txt"**, that user will provide will have the following format :

```
5
2
4
5
6
7
```

In the file above, we define a vector with 5 elements (the first number), and provide 5 elements of the vector.

The code below reads the input file, calculates the average, and writes that average value to the "output.txt" file.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    /* declare variables */
    int i;
    int length; // # of elements in the vector (read from file)
    FILE *fptr_in;  // input file pointer
    FILE *fptr_out; // output file pointer
    int *vector; // declaration for the vector. Size will be determined later.
    int sum;
    float avg;

    /* open the input and output files first */
```

```
    /* input file */
    fptr_in = fopen("vec_in.txt","r");

    /* output file */
    fptr_out = fopen("output.txt","w");

    /* we know that the first number written in the file is the length of the vector */
    fscanf(fptr_in, "%d", &length);

    /* Now we know how many elements we will read form the file and store in our vector */
    /* allocate memory for the vector */
    vector = (int *) malloc(length * sizeof(int));

     /* read length number of elements from the file and store in the vector  */
     for (i = 0; i < length; i++)
     {
         fscanf(fptr_in, "%d", &vector[i]);
     }

     /* first, calculate the sum of all elements */
     sum = 0;

     for (i = 0; i < length; i++)
     {
         sum += vector[i]; /* sum = sum + vector[i]; */
     }

     /* calculate the average */
     avg = (float) (sum / length);

     /* write the average value to the output file */
     fprintf(fptr_out,"%f",avg);

     /* close the input file */
     fclose(fptr_in);

    /* close the output file */
    fclose(fptr_out);

    return 0;
}
```

Functions in C The last thing we will learn in the section is the use of C functions. A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions and increase the code reusability.

Similar to variables, we need to declare functions before we can use them. A function decla-

ration tells the compiler about a function's name, return type, and input parameters. After the function declaration, we write the function definition, which is the actual boy of the function.

Function definition The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter list )
{
    body of the function
}
```

Below, you can see an example function that takes two numbers and returns the maximum value between the two:

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function declaration A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has thw following parts

```
return_type function_name( parameter list );
```

Below, you can see the function declaration for the max function we have written above:

```
int max(int num1, int num2);
```

Calling a function While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

For example

```c
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int maxval;

   /* calling a function to get max value */
   maxval = max(a, b);

   printf( "Max value is : %d\n", maxval );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

An even more improved version of the vector average program Last time we added some functionality to our vector average program. Since calculating the average value of a vector is a commom operator, you may want to perform this task in another program as well. Instead of copy-pasting your previous code, it makes more sense to write a function that takes a vector and returns the average value of that vector. Written this way, you can use the same code over and over again in other program without any changes.

Below is the new version of the vector addition program:

```c
#include <stdio.h>
#include <stdlib.h>

/* declare the average function here
 * input parameters are the integer vector and its length
 * output will be the average value in floating point format
```

```
 */

float calculate_average(int *vec, int length);

int main()
{
   /* declare variables */
   int i;
   int length; // # of elements in the vector (read from file)
   FILE *fptr_in;  // input file pointer
   FILE *fptr_out; // output file pointer
   int *vector; // declaration for the vector. Size will be determined later.
   float avg;

   /* open the input and output files first */
   /* input file */
   fptr_in = fopen("vec_in.txt","r");

   /* output file */
   fptr_out = fopen("output.txt","w");

   /* we know that the first number written in the file is the length of the vector */
   fscanf(fptr_in, "%d", &length);

   /* Now we know how many elements we will read form the file and store in our vector */
   /* allocate memory for the vector */
   vector = (int *) malloc(length * sizeof(int));

    /* read length number of elements from the file and store in the vector  */
    for (i = 0; i < length; i++)
    {
        fscanf(fptr_in, "%d", &vector[i]);
    }

    /* calculate the average value of the vector */
    avg = calculate_average(vector, length);

    /* write the average value to the output file */
    fprintf(fptr_out,"%f",avg);

    /* close the input file */
    fclose(fptr_in);

   /* close the output file */
   fclose(fptr_out);

   return 0;
}
```

```
float calculate_average(int *vec, int length){
    int i;
    int sum;
    float average;

   /* first, calculate the sum of all elements */
    sum = 0;

    for (i = 0; i < length; i++)
    {
        sum += vec[i]; /* sum = sum + vector[i]; */
    }

    /* calculate the average */
    average = (float) (sum / length);

    /* this function, by definiton, returns a float value */
    // return the average value to the main program

    return average;
}
```

0.10.6 Madagascar API

So far, you, have learned a lot about the C programming language. Knowledge you have should allow you to write many different programs. However, as you develop more and more programs you will realize there will be a lot overlap between the programs and you will be writing the same code over and over again. In such cases, it is crucial to have a standard library for specific tasks such as I/O to improve the code reusability.

Madagascar provides interfaces for programming in C and optionally in C++ and Fortran90, etc. Programming using a language for which an API (Applications Programming Interface) is provided by Madagascar allows the user to operate on RSF files and also use some predefined functions from the RSF library. These standardized file format and predefined function helps developers avoid spending time on those opearations, but instead focus in their research. From now on, we will develop our applications using the Madagascar API and see that we can do everything we have learned so far using this API, in most cases, in more efficient ways.

Below, is our vector average program written using the Madagascar API. Since the standard file format for Madagascar is RSF, the program reads the vector from an **rsf** file and write the final average value to an output **rsf** file.

```
#include <rsf.h> /* include the Madagascar header */
#include <stdio.h>
#include <stdlib.h>

/* declare the average function here
 * input parameters are the integer vector and its length
```

```
 * output will be the average value in floating point format
 */

float calculate_average(float *vec, int length);

int main(int argc, char* argv[])
{
   /* declare variables */
   int i;
   int length; // # of elements in the vector (read from file)
   sf_file file_in, file_out;  // input & output RSF files
   float *vector; // declaration for the vector. Size will be determined later.
   float avg;

   /* input and output axes */
   sf_axis ai, ao;

    /* Initialize RSF */
    sf_init(argc,argv);

   /* open the input and output files first */
   /* input RSF file */
   file_in = sf_input("in"); /* input file */

   /* output RSF file */
   file_out = sf_output("out"); /* output file */

   /* we can read the length of the vector from the RSF file header */
   /* input the first axis of the input file */
   ai = sf_iaxa(file_in,1);

   /* length of the vector */
   length = sf_n(ai);

   /* write the length of the vector */
   sf_warning("length of the input vector %d",length);

   /* make the output file axis */
   ao = sf_maxa(1,0,0);

   /* assign the output axis to the output file */
   sf_oaxa(file_out, ao, 1);

   /* Now we know how many elements we will read form the file and store in our vector */
   /* Madagascar provides functionality for dynamic memory allocation */
   vector = sf_floatalloc(length);

   /* read the RSF file */
```

```
    sf_floatread(vector, length, file_in);

    /* calculate the average value of the vector */
    avg = calculate_average(vector, length);

    /* write the average value to the terminal */
    sf_warning("average value: %f",avg);

     /* write the average value to the output file */
     sf_floatwrite(&avg, 1, file_out);


     sf_close();
     exit(0)
}

float calculate_average(float *vec, int length){
     int i;
     float sum;
     float average;

    /* first, calculate the sum of all elements */
     sum = 0;

     for (i = 0; i < length; i++)
     {
         sum += vec[i]; /* sum = sum + vector[i]; */
     }

     /* calculate the average */
     average = sum / length;

     /* this function, by definiton, returns a float value */
     // return the average value to the main program

     return average;
}
```

Let us examine the above program in detail.

```
#include <rsf.h>
```

The include preprocessing directive is required to access the RSF interface

```
sf_file file_in, file_out;  // input & output RSF files
```

RSF data files are defined with an abstract sf_file data type. An abstract data type means that the contents of it are not publicly declared, and all operations on sf_file objects should be performed with library functions. This is analogous to FILE * data type used in stdio.h

```
/* input and output axes */
sf_axis ai, ao;
```

RSF files have headers that contain information about the dimensions of the file and the sampling of the each axis. When using the Madagascar API, we retrieve that information from the input file to be used further. Similarly, we define the axes of the output files in our code to have the correct headers. Here, sf_axis is a simple C structure that keeps the number of elements along the axis, sample spacing and the origin of that axis.

```
/* Initialize RSF */
sf_init(argc,argv);
```

Before using any of the other functions, you must call sf_init. This function parses the command line and initializes an internally stored table of command-line parameters.

```
/* input RSF file */
file_in = sf_input("in"); /* input file */

/* output RSF file */
file_out = sf_output("out"); /* output file */
```

The input and output RSF file objects are created with sf_input and sf_output constructor functions. Both these functions take a string argument. The string may refer to a file name or a file tag. For example, if the command line contains vel=velocity.rsf, then both sf_input("velocity.rsf") and sf_input("vel") are acceptable. Two tags are special: "in" refers to the file in the standard input and "out" refers to the file in the standard output.

```
/* input the first axis of the input file */
ai = sf_iaxa(file_in,1);
```

sf_iaxa is a Madagascar function that reads the axis from an input RSF file. Here, we read the first axis of the file and keep that in the sf_axis structure "ai".

```
/* make the output file axis */
ao = sf_maxa(1,0,0);
```

It is important to have the correct headers in your output files. To ensure that, we define axes inside our. **sf_maxa** is an **axis making** function that takes three parameters: **number of parameters along the axis, sample spacing, origin.**

```
/* assign the output axis to the output file */
sf_oaxa(file_out, ao, 1);
```

We define the first axis of the outpuf file (the last parameter) by assigning the axis we made earlier to the first dimension of the output file.

```
vector = sf_floatalloc(length);
```

Next, we allocate an array of floating-point numbers to store a trace with the library sf_floatalloc function. Unlike the standard malloc the RSF allocation function checks for errors and either terminates the program or returns a valid pointer.

```
sf_floatread(vector, length, file_in);
sf_floatwrite(&avg, 1, file_out);
```

sf_floatread & sf_floatwrite are Madagascar API functions that reads/writes float values from/to rsf files.

```
sf_warning("average value: %f",avg);
```

You can use the sf_warning function instead of printf when using the Madagascar API.

```
sf_close();
exit(0)
```

We explicitly close the input file to avoid leaving a stale temporary file in $DATAPATH if the program is called in a pipe sequence. Then, we close the program by sending the shell the Unix code that tells it no errors were encountered.

0.10.7 To compile

1. Name your file as Mvecavg.c
2. Put your code into your user directory in $RSFSRC/user/myname
3. Open your SConstruct in the same directory and add the "vecavg" to targets.c list
4. in that directory, scons sfvecavg
5. generate a vector in Magadascar (e.g., sfspike) and provide it as an input to the "sfvecavg" program

```
In [ ]:
```