

# Exercises 1: Getting started

NMNV565: High-Performance Computing for  
Computational Science

# Today's Tasks

---

1. Logging on to local cluster
  - Properties of the local cluster
2. Terminal/command line basics
  - Navigating file system
  - Editing a file
  - Moving files between machines
3. A brief C tutorial
  - Helloworld program
  - Simple Addition program
  - Compiling and running a C program
  - SLURM job submissions system
  - Further online tutorials...

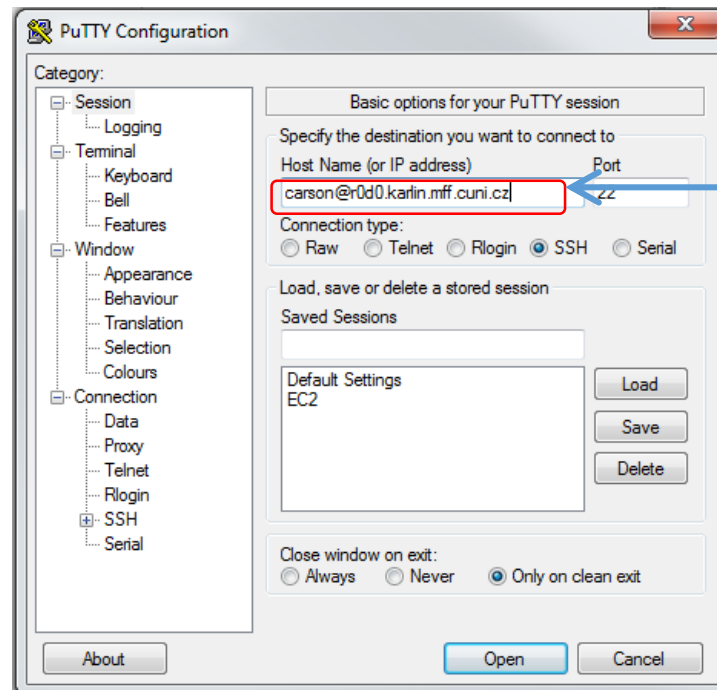
# The Karlin Cluster

# Logging on

- In Linux/Mac, open the Terminal program
  - Type

```
ssh yourusername@r3d3.karlin.mff.cuni.cz
```

- In Windows, you will need to download the program PuTTY:  
<https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>



In this box, put  
yourusername@r3d3.  
karlin.mff.cuni.cz

# Logging on

- You will then be asked to enter your password.
- If it is your first time logging on, you will be asked if you'd like to add the hostname to the list of hosts. Click "yes".

<http://cluster.karlin.mff.cuni.cz/pouziti-clusteru/zaklady-vzdaleneho-pristupu/>

```
r0d0.karlin.mff.cuni.cz - PuTTY
Welcome to cluster SNEHURKA (Ubuntu 16.04.6 LTS GNU/Linux 4.13.0-37-generic x86_64)

System information as of Mon Sep 30 11:34:51 CEST 2019

System load:  0.0          Users logged in:      4
Usage of /:   32.0% of 67.17GB  IP address for eth0:  192.168.11.171
Memory usage: 67%          IP address for eth1:  195.113.30.26
Swap usage:   0%           IP address for docker0: 172.17.0.1
Processes:   315

=> There are 5 zombie processes.

Graph this data and manage this system at:
https://landscape.canonical.com/

35 packages can be updated.
24 updates are security updates.

Last login: Mon Sep 30 11:31:12 2019 from 10.113.3.219

Vitejte na clusteru Snehurka

* Informace o obsazenosti clusteru: freenodes
* Login nody: r0d0 r1d1
* Nove verze modulu: paraview/5.6.0 python/3.6.5 python/3.6.5-ompi fenics/2017.2
.0-ompi
* Nove moduly: julia/1.0.3

Pokud narazite na problem, nebo nefucni program, prosime piste
na adresu clusteradmin@karlin.mff.cuni.cz

!***! 21.1.2019 BYL PREUSPORADAN DATOVY PROSTOR CLUSTERU

!***! Adresarove struktury a jejich dostupnost:
!***! - DAS - Domovsky Adresar Sekce (posta, clanky,...) - zalohovane (obsahuje
data vaseho domovskeho adresare)
!***! - DAC - Domovsky Adresar Clusteru (zdrojove kody, vstupni data uloh,...) -
zalohovane
!***! - PAW - Pracovni Adresar Clusteru (behova data vypoctu,...) - nezalohovane
(obsahuje byvala data /usr/nobackup)
!***! |-----cluster-----| |-----sekce-----|
!***! r0d0,r1d1          r1,...,r28          hill,jarre          windows
!***! DAS /srv/groot/login nedostupny      /usr/users/login   M:(\\homes\log
in)
!***! DAC /usr/users/login /usr/users/login /usr/cluster/login V:(\\hrablo\cl
uster-home)
!***! PAC /usr/work/login /usr/work/login /usr/work/login W:(\\hrablo\cl
uster-work)
carson@r0d0:~$
```

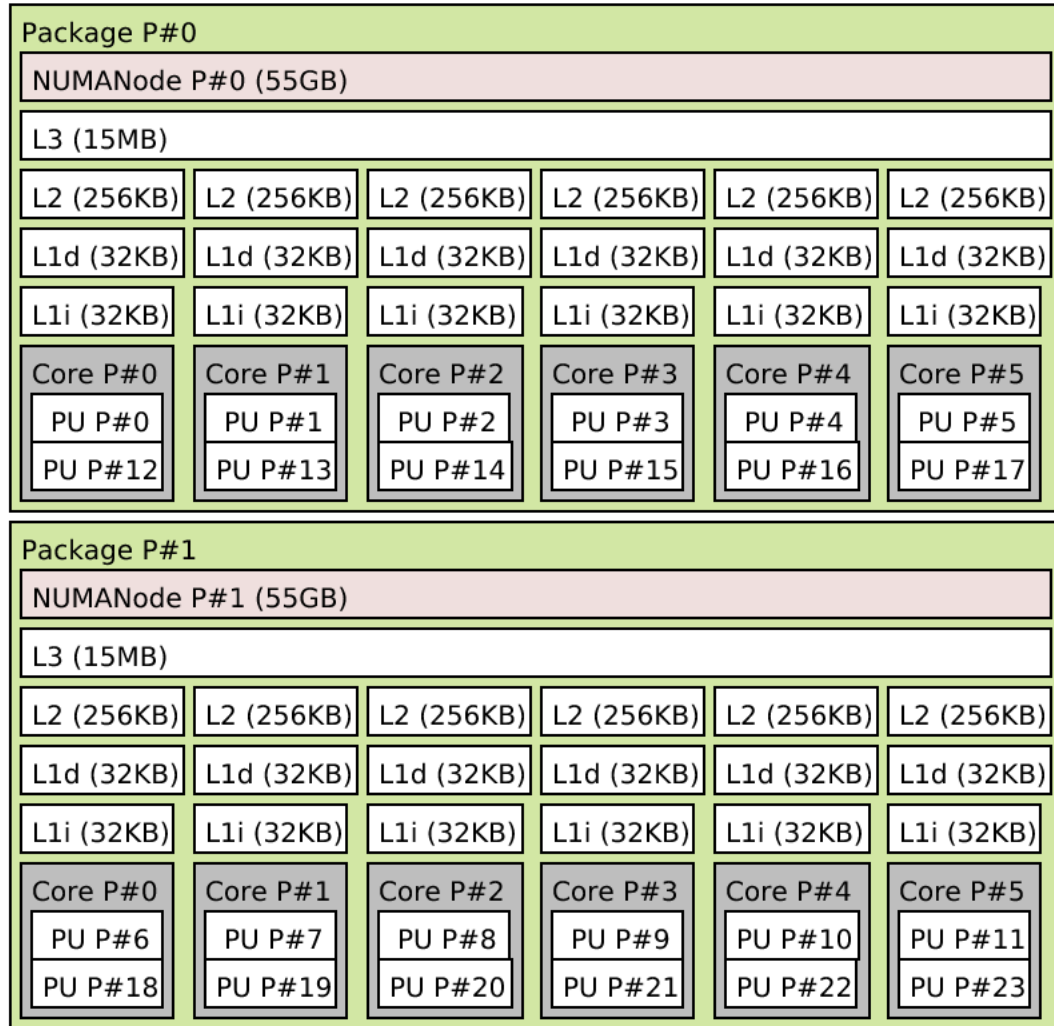
# The Karlin Cluster

- Hardware

- The main access point is called r3d3.karlin.mff.cuni.cz
- nodes r3-r5 are Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz, 32GB RAM
- node r6 is Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz, 115GB RAM
- nodes r21-r27 are Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz, 64GB RAM
- nodes r1-r7, r21-r27 connected by InfiniBand with a capacity of 40 Gb / sec
- nodes r31-r40 connected by InfiniBand with a capacity of 100 Gb / sec
- nodes r31-r35 are 2x Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz, 132GB RAM
- nodes r36-r39 are 2x Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz, 132GB RAM
- node r40 is 4x Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz, 512GB RAM
- node g1 is 2x Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz, 64GB RAM, GeForce RTX 2080 Ti Rev. A
- node g2 is 2x Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz, 64GB RAM, 2x GeForce RTX 2080 Ti Rev. A

<http://cluster.karlin.mff.cuni.cz/snehurka/hardware/>

Machine (110GB total)



Host: r6

# Hardware

---

- Type

freenodes

to see what is running on the cluster



# Command Line Tutorial

# Command Line

---

- Goal: Learn what is the command line? What is a terminal?
- Resources:
  - <https://tutorials.ubuntu.com/tutorial/command-line-for-beginners>

# Location and Working Directory

- When you are in the terminal, you are sitting in some folder in the file system. When you issue commands, you are issuing them from that folder. This is called the working directory.

`pwd` (print **w**orking **d**irectory)

- You can create a new folder with the command "mkdir". Try it:

`mkdir exercises1`

- You can see the files and other folders in your current working directory by typing

`ls` (list)

- You can change the working directory using the command

`cd exercises1` (change working **d**irectory to newly created folder)

# Location and Working Directory

---

- Type

```
cd ..
```

Where are you now? Type `pwd` to find out.

The two dots `..` is a shortcut to the parent directory

Go back to the `exercises1` directory (`cd exercises1`)

# Creating files and folders

- Create a few subdirectories:

```
mkdir dir1 dir2 dir3
```

- So far we've only seen commands that work on their own (cd, pwd) or that have a single item afterwards. But this time we've added three things after the mkdir command. Those things are referred to as ***parameters or arguments***, and different commands can accept different numbers of arguments. The mkdir command expects at least one argument, whereas the cd command can work with zero or one, but no more.
- See what happens when you try to pass the wrong number of parameters to a command:

```
mkdir
```

```
cd /etc ~/Desktop
```

# Creating files using redirection

- First, remind yourself what the `ls` command is currently showing:

```
ls
```

- Suppose we wanted to capture the output of that command into a text file. To do this, add the greater-than character ("`>`") to the end of our command line, followed by the name of the file to write to:

```
ls > output.txt
```

- This time there's nothing printed to the screen. The output is being redirected to our file instead.
- Run `ls` again and you should see that the `output.txt` file has been created.
- We can use the `cat` command to look at its content:

```
cat output.txt
```

# Creating files using redirection

- Let's look at another command, echo:

```
echo "This is a test"
```

- `echo` just prints its arguments back out again.
- If you combine it with a redirect, you have a way to easily create small test files:

```
echo "This is a test" > test_1.txt
```

```
echo "This is a second test" > test_2.txt
```

```
echo "This is a third test" > test_3.txt
```

```
ls
```

# Concatenation of Files

---

- You should `cat` each of these files to check their contents.
- Note that `cat` is more than just a file viewer - its name comes from 'concatenate', meaning "to link together".
- If you pass more than one filename to `cat` it will output each of them, one after the other, as a single block of text:

```
cat test_1.txt test_2.txt test_3.txt
```



# Editing a File

---

- nano
- vi
- emacs

# Copying a file

- To copy a file, use the `cp` command.

- The syntax is

```
cp [OPTION] Source Destination
```

```
cp [OPTION] Source Directory
```

```
cp [OPTION] Source-1 Source-2 Source-3 Source-n Directory
```

- Notice that the second argument can either be the name of a file or a directory

- What will the following commands do?

```
mkdir tmp
```

```
cp test_3.txt text_3_copy.txt
```

```
cp test_3.txt tmp
```

```
cp test_3.txt tmp/text_3_copy.txt
```

```
cp test_2.txt test_3.txt tmp
```

# Moving a file

- The command `mv` can be used for moving (or renaming files)
- Syntax:

```
mv [options] source dest
```

- What will the following do?

```
mv test_1.txt test_1_renamed.txt
```

```
mv test_1.txt tmp
```

```
mv test*.txt tmp
```

# Deleting a File

---

- The command for deleting files is `rm`
- To delete a file, e.g.,

```
rm test_1.txt
```

- Try deleting a directory with

```
rm tmp
```

- What happens?
- You need to use the `-r` option to recursively delete the whole folder and its contents:

```
rm -r tmp
```

# Deleting a File

## Important Warning!

- Unlike graphical interfaces, `rm` doesn't move files to a folder called "trash" or similar.
  - It **deletes them totally, utterly and irrevocably!**
  - You need to be *very* careful with the parameters you use with `rm` to make sure you're only deleting the file(s) you intend to.
- Be especially careful when using wildcards, as it's easy to accidentally delete more files than you intended.
- An errant space character in your command can change it completely:

```
rm t*
```

means "delete all the files starting with t", whereas

```
rm t *
```

means "delete the file t as well as any file whose name consists of zero or more characters — which would be **everything in the directory!**"

- **If you're at all uncertain use the `-i` (interactive) option to `rm`**, which will prompt you to confirm the deletion of each file; enter Y to delete it, N to keep it, and press Ctrl-C to stop the operation entirely.

# Moving a file to your local machine

Create a file called `cpuinfo.txt` which has the `cpuinfo` by typing

```
cat /proc/cpuinfo > cpuinfo.txt
```

To move this file to your local machine, you need to work from a directory on your local machine! (Open new terminal window in Linux/Max, or on Windows, install `pscp` and run cmd from the directory where you have `pscp.exe`)

<https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>

General format: `scp [source] [destination]`

```
scp carson@r3d3.karlin.mff.cuni.cz:/usr/users/carson/exercises1/cpuinfo.txt cpuinfo.txt
```

```
pscp -scp carson@r3d3.karlin.mff.cuni.cz:/usr/users/carson/exercises1/cpuinfo.txt cpuinfo.txt
```

# Moving a file to the cluster

---

- You should download the ex1.tar file from the Moodle site
- We want to move this file to your directory on the cluster
- Remember: scp [source] [destination]

```
scp ex1.tar carson@r3d3.karlin.mff.cuni.cz:/usr/users/carson/exercises1/.
```

```
pscp -scp ex1.tar carson@r3d3.karlin.mff.cuni.cz:/usr/users/carson/exercises1/.
```

# File Compression/Decompression

---

- The tar program stores and extracts files from an archive
- To extract files from the ex1.tar file, we want to use the command

```
tar -xf ex1.tar
```

If you type `ls`, you should now be able to see the two files that were in `ex1.tar`, `helloworld.c` and `sum.c`



# Manual pages

---

- If you are having trouble using any command line programs, there are built-in manuals that you can use for guidance. Just type "man" + the name of the command/program
- Examples:
  - man scp
  - man tar
  - man ls

# Other Resources

---

- The Linux Command Line, A Book by William Shotts (Free PDF download) <http://linuxcommand.org/tlcl.php>
- There are many Command Line cheat sheets! (Google "linux command line cheat sheet" and pick your favorite to print and hang above your desk)

# C Tutorial

# Why C?

- C is a general purpose programming language, which relates closely to the way machines work.
- Gives us an understanding how computer memory works, fine grained control over computations and memory management
- C is very commonly used - it is the language of many applications (Windows, the Python interpreter, Git, etc).
- C is high-level enough to allow us to write and manipulate code easily, but it is low-level enough to be close to the hardware, allowing us finer-grained control over how our algorithms perform
  - Understanding the "efficiency layer" is important, even if we intend to work at the "productivity layer"!

# Your first C program

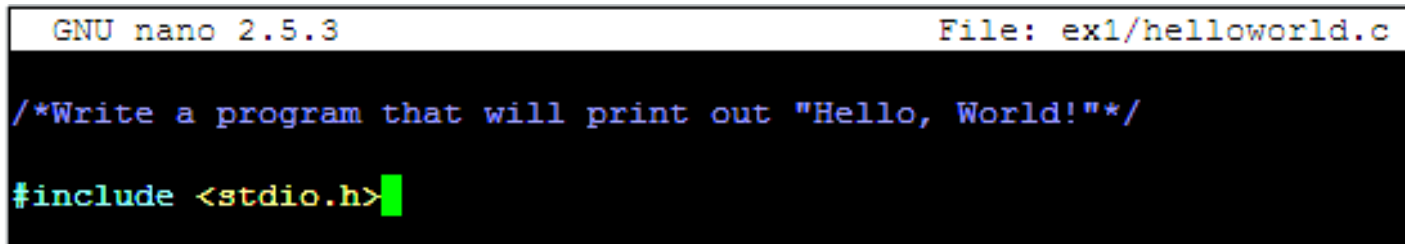
- Open helloworld.c in an editor (nano, vi, emacs)
- It should be blank except for comments

```
GNU nano 2.5.3                               File: ex1/helloworld.c
/*Write a program that will print out "Hello, World!"*/
█
```

[https://www.learn-c.org/en/Hello%2C\\_World%21](https://www.learn-c.org/en/Hello%2C_World%21)

# Your first C program

- Every C program uses libraries, which give the ability to execute necessary functions. For example, the most basic function called `printf`, which prints to the screen, is defined in the `stdio.h` header file.
- To add the ability to run the `printf` command to our program, we must add the following include directive to our first line of the code:



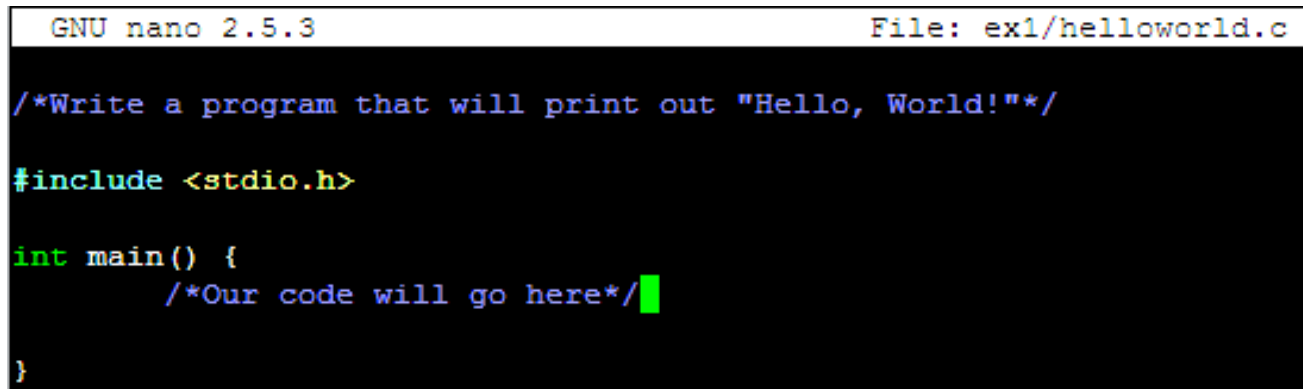
```
GNU nano 2.5.3 File: ex1/helloworld.c

/*Write a program that will print out "Hello, World!"*/
#include <stdio.h>
```

This is a preprocessor command. That notifies the compiler to include the header file `stdio.h` in the program before compiling the source-code.

# Your first C program

- The second part of the code is the actual code which we are going to write. The first code which will run will always reside in the main function.

A screenshot of a GNU nano 2.5.3 text editor window. The title bar at the top shows "GNU nano 2.5.3" on the left and "File: ex1/helloworld.c" on the right. The editor area has a black background with syntax-highlighted C code. The code includes a multi-line comment, an include directive for <stdio.h>, and the start of a main function with curly braces and a comment indicating where the user's code should go.

```
GNU nano 2.5.3                               File: ex1/helloworld.c

/*Write a program that will print out "Hello, World!"*/

#include <stdio.h>

int main() {
    /*Our code will go here*/
}
```

- The int keyword indicates that the function main will return an integer - a simple number.
- The main() is the main function where program execution begins. Every C program must contain only one main function.
- Curly braces are used to bound the scope of the main() function

# Your first C program

- For this tutorial, we will return 0 to indicate that our program was successful:

```
GNU nano 2.5.3 File: ex1/helloworld.c

/*Write a program that will print out "Hello, World!"*/

#include <stdio.h>

int main() {
    /*Our code will go here*/

    return 0;
}
```

- Notice that every line in C must end with a semicolon, so that the compiler knows that a new line has started.
- Last but not least, we will need to call the function printf to print our sentence.

```
GNU nano 2.5.3 File: ex1/helloworld.c

/*Write a program that will print out "Hello, World!"*/


#include <stdio.h>

int main() {
    /*Our code will go here*/
    printf("Hello, World!\n");
    return 0;
}
```



# General Structure of a C Program

```
//Name of program  
//Author
```



Documentation section

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



Preprocessor directives

```
#define max 100
```




Definition section

```
void myfunc();  
int x = 100;
```




Global declaration section

```
int main(){  
    int a = 100;  
    myfunc();  
    return 0;  
}
```



Main function

```
void myfunc(){  
    printf("%d\n",max);  
}
```



Function definition

# Your first C program

---

- Save the file
- We now need to compile the C code into a program that can be run
- For this we will use the gcc tool (GNU Compiler Collection)

```
gcc -c helloworld.c
```

This compiles the code into an object file named `helloworld.o`

Now, we have to tell the linker to take the object file and make it into an executable program:

```
gcc -o helloworld helloworld.o
```

You can do both of these in one step if you prefer:

```
gcc -o helloworld helloworld.c
```

# Your first C program

---

- Now run the program. Type  
`./helloworld`

# Data Types and Variables

- C has several types of variables, but there are a few basic types:
  - Integers - whole numbers which can be either positive or negative. Defined using char (1 byte), int (4 bytes), short (2 bytes), long (8 bytes)
    - Can be signed or unsigned
  - Floating point numbers - real numbers (numbers with fractions). Defined using float (4 bytes) and double (8 bytes)
  - Structures - user-defined grouped list of variables. Example:

```
struct point {  
    int    x;  
    int    y;  
};
```

# Data Types and Variables

- Note that C does not have a boolean type. Usually, it is defined using the following notation:

```
#define BOOL char  
#define FALSE 0  
#define TRUE 1
```

- C uses arrays of characters to define strings, and will be explained later

# Defining Variables

- To define the variables foo and bar, we need to use the following syntax:

```
int foo;  
int bar = 1;
```

- The variable foo can be used, but since we did not initialize it, we don't know what's in it. The variable bar contains the number 1.
- Now, we can do some math. Assuming a, b, c, d, and e are variables, we can simply use plus, minus and multiplication operators in the following notation, and assign a new value to a:

```
int a = 0, b = 1, c = 2, d = 3, e = 4;  
a = b - c + d * e;  
printf("%d\n", a); /*will print 1-2+3*4=11*/
```

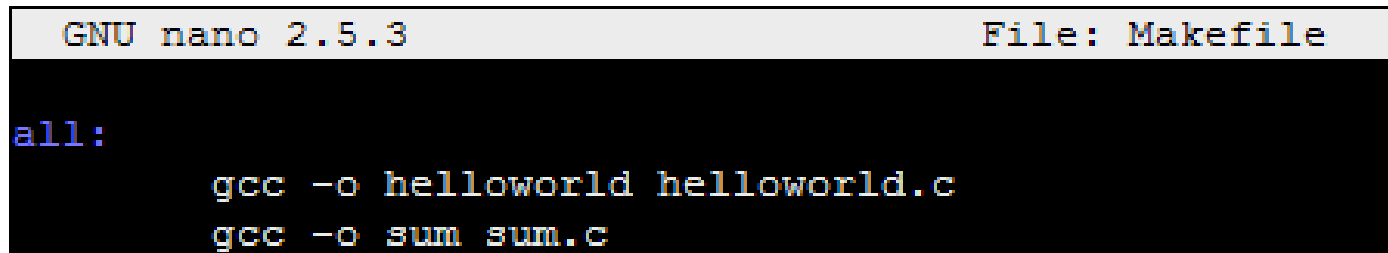
# Your Second C Program

- Open the file `sum.c` for editing
- Create a variable `sum` which gives the sum of the defined variables `a`, `b`, and `c`.
- Close the file and compile the code into an executable program
- Run the program from the command line

```
carson@r0d0:[~/exercises1/ex1]: ./sum  
The sum of a, b, and c is 12.750000.
```

# The Makefile

- A makefile is a file that contains instructions on how to compile and link a program
- When program has many build/link dependencies, the makefile is an easy place to keep them all organized. Compilation and linking can then be done with a simple command
- Create a file called `Makefile` and open it for editing:



```
GNU nano 2.5.3 File: Makefile

all:
    gcc -o helloworld helloworld.c
    gcc -o sum sum.c
```

In the command line, type:

```
make all
```

This will compile and link the two programs you just wrote.



# The Makefile

- A slightly more complicated version of the Makefile:

Try it.

What happens when you  
type :

make clean

make helloworld

make sum

make all

```
GNU nano 2.5.3                                     File: Makefile

all: helloworld sum

helloworld: helloworld.o
        gcc -o helloworld helloworld.o

helloworld.o: helloworld.c
        gcc -c helloworld.c

sum: sum.o
        gcc -o sum sum.o

sum.o: sum.c
        gcc -c sum.c

clean:
        rm -rf *.o helloworld sum
```

# Submitting a job to the cluster

---

- You can submit a job to the cluster using SLURM
- For detailed instructions, see <https://cluster.karlin.mff.cuni.cz/pouziti-clusteru/spravce-uloh-slurm/>

# SLURM commands

- **sacct**: display accounting data for all jobs and job steps in the Slurm database
- **sacctmgr**: display and modify Slurm account information
- **salloc**: request an interactive job allocation
- **sattach**: attach to a running job step
- **sbatch**: submit a batch script to Slurm
- **scancel**: cancel a job or job step or signal a running job or job step
- **scontrol**: display (and modify when permitted) the status of Slurm entities. Entities include: jobs, job steps, nodes, partitions, reservations, etc.
- **sdiag**: display scheduling statistics and timing parameters
- **sinfo**: display node partition (queue) summary information
- **smap**: a curses-based tool for displaying jobs, partitions, reservations, and Blue Gene blocks
- **sprio**: display the factors that comprise a job's scheduling priority
- **squeue**: display the jobs in the scheduling queues, one job per line
- **sreport**: generate canned reports from job accounting data and machine utilization statistics
- **srund**: launch one or more tasks of an application across requested resources
- **sshare**: display the shares and usage for each charge account and user
- **sstat**: display process statistics of a running job step
- **sview**: a graphical tool for displaying jobs, partitions, reservations, and Blue Gene blocks

- <http://cluster.karlin.mff.cuni.cz/pouziti-clusteru/spravce-uloh-slurm/>

# Create a simple job script

Use nano (or vi, emacs) to create a file called `job.sh` with the following contents:

```
#!/bin/bash
#SBATCH --job-name=HW1
#SBATCH --output=hw_output.txt
#SBATCH -N 1
#SBATCH -p short

srun ./helloworld
~
~
```

Type

```
sbatch job.sh
```

to submit the job into the queuing system

Once the job is finished, you should see the file `hw_output.txt` in your working directory (type `ls` to check if it is there yet)

# Create a simple job script

---

- This was a short job, so it probably ran right away.
- Longer jobs might have to wait in the queue longer
- To check the status of your job, you can use the command `squeue`

```
squeue -j jobid
```

```
squeue -u username
```

- Type

```
cat hw_output.txt
```

to see the output of your job

# Pointers in C

- When a variable gets declared, memory to hold a variable of that type is allocated at an unused memory location
- The location that is allocated is the variable's memory address
- For a compiler, a variable is a symbol for a starting memory address
  - *To a compiler all variables are just memory addresses and sizes*
- An int holds an integer number, a float holds a floating point decimal number. **A pointer is a variable that holds the memory address of another variable.**

# Pointers in C

- Two main operators for working with pointers
- The **\* operator**: used when declaring a pointer and when "dereferencing" a pointer (gives the value stored in a pointer)
- The **& operator**: used to get the address of another variable. It is used to assign a value to a pointer.
  - Putting the & operator in front of another variable returns a pointer to that variable of the type of that variable

```
#include <stdio.h>
int main(){
    int* ptr; —————> declare an int pointer name ptr
    int val = 1; —————> declare an int with the value of 1

    ptr = &val; —————> get the address of the val variable and
    printf("ptr = &val = %p\n", ptr); store it in ptr

    int deref = *ptr; —————> dereference the ptr variable to get
    printf("deref = *ptr = %d\n", deref); the int value at the address stored

    *ptr = 2; —————> dereference the ptr variable to set
    printf("val = %d\n", val); the int value at the address stored
}
```

# Memory Allocation in C

- In C, you need to manage your own memory

- To allocate memory: `malloc()`

```
void* malloc (size_t size)
```

- size is size of memory block in bytes
- returns a pointer to the allocated memory

- Example:

```
double* myarray = (double*) malloc(10*sizeof(double));
```

- the `(double *)` is called a cast

- To free memory: `free()`

- Example:

```
free(myarray);
```



# Malloc vs. Calloc

- `void *malloc(size_t n)`
  - returns a pointer to `n` bytes of uninitialized storage, or `NULL` if the request cannot be satisfied
  - takes one argument that is, *number of bytes*.
  - Doesn't initialize memory entries
- `void *calloc(size_t n, size_t size)`
  - returns a pointer to enough free space for an array of `n` objects of the specified size, or `NULL` if the request cannot be satisfied.
  - takes two arguments those are: *number of blocks* and *size of each block*.
  - Initializes memory entries to 0

# Rest of the Session

---

- Continue working through the C tutorial exercises, starting here:  
<https://www.learn-c.org/en/Arrays>
- You can write code and do the examples in-browser
- Will guide you through pointers, memory allocation