

Session #3 Part 1

Text manipulation, I/O and redirection & Permissions



Summary

1. CLI Text Editors (Nano)
2. Reading Files
 - 2.1. cat
 - 2.2. grep
3. Using g++ to compile C++ files
4. Input and Output Streams and Redirection
 - 4.1. Input and Output Streams
 - 4.2. I/O Redirection
 - 4.3. Piping ' | '
5. File Permissions
 - 5.1. What permissions are
 - 5.2. Changing Permissions
 - 5.3. Changing Ownership

CLI Text Editors

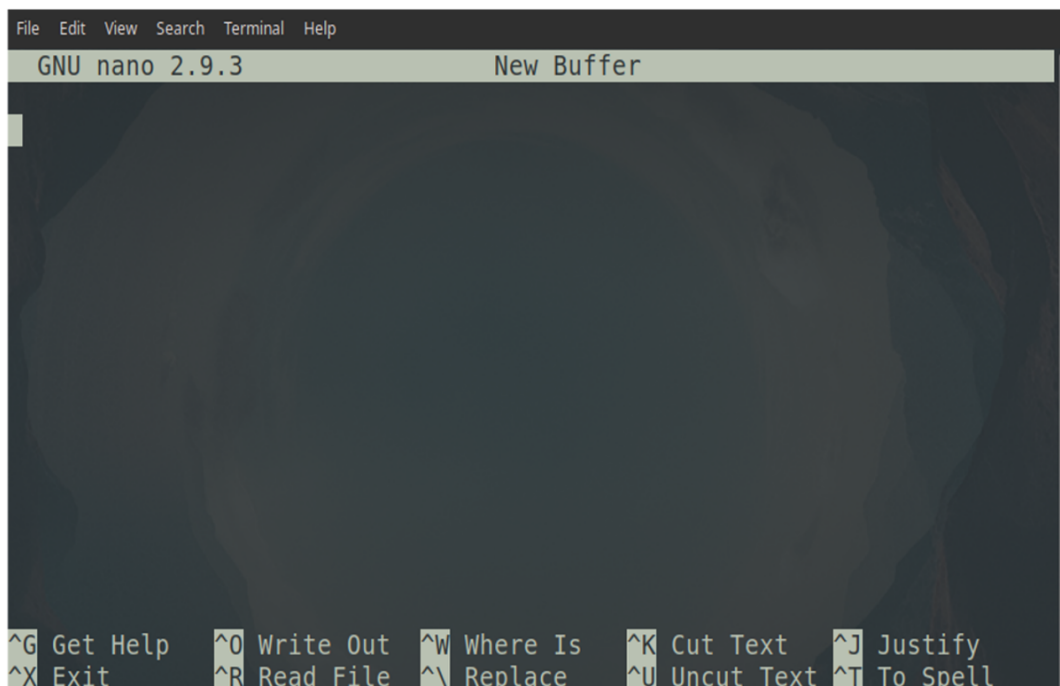
GNU Nano

GNU Nano or Nano for short, is a text editor for Unix-like computing systems or operating environments using a command line interface(CLI).

I.e Linux, Unix, macOS, etc..

To start using nano we can simply run it from the terminal by running the command `nano`.

We can write whatever text we want and once we're done we can exit using the shortcut Ctrl + x, we'll be prompted to enter the name of the file to save as.



Another way to open nano is to write nano in the terminal followed by the file name or the absolute path of the file we want to edit and if the file doesn't exist it will be created.

Example: `nano /etc/fstab`

or `nano file.txt`

Reading Files

cat

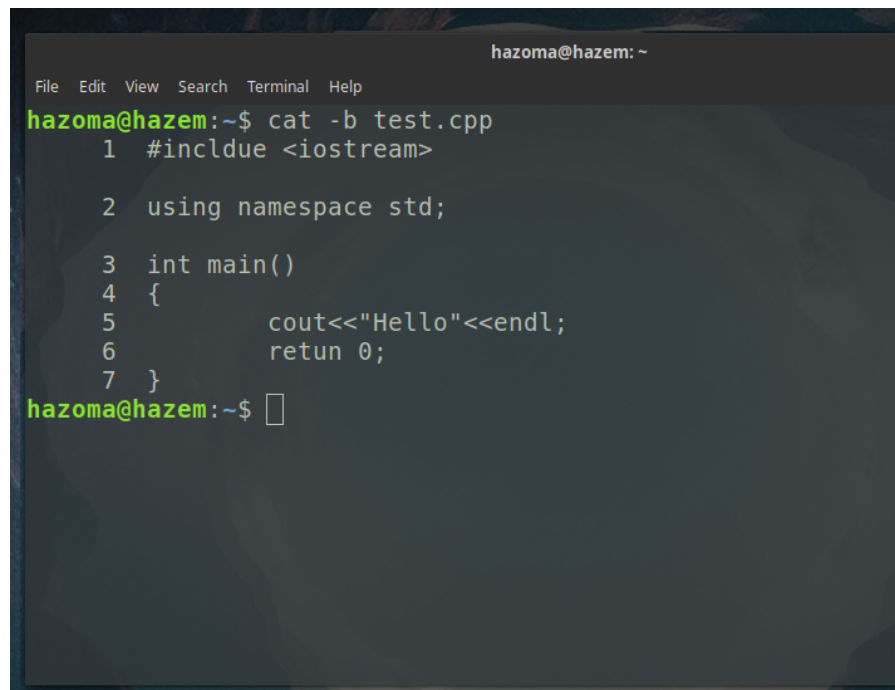
cat is a basic Linux command used to concatenate files and print output to the standard output.

We'll get to concatenating files later, for now we can use cat to print the content of a file we can type cat followed by the file.

We can use some options to format the output, we can use:

-n to number the lines

-b to only number lines containing text like the example shown in the photo.



```
hazoma@hazem: ~  
File Edit View Search Terminal Help  
hazoma@hazem:~$ cat -b test.cpp  
 1  #include <iostream>  
  
 2  using namespace std;  
  
 3  int main()  
 4  {  
 5      cout<<"Hello"<<endl;  
 6      return 0;  
 7  }  
hazoma@hazem:~$
```

To concatenate files using cat, just add more arguments to the command:

cat file1 file2 file3

grep

grep is a Linux utility used to find patterns in files.

grep is used to search in files, we type **grep** in the terminal followed by the string/pattern/expression we want to find then the file we want to search in and we can also type multiple files to search in.

Example:

```
grep "Pattern" File1 /home/user/Desktop/File2
```

The previous example searches for the phrase “Pattern” in File1 and File2.

Commonly used options with grep:

-i : Ignores the case of the pattern we’re searching for.

Example: Searching for the string PATTERN in a file containing the word pattern will be considered a match.

-c : Counts the number of lines which contain the pattern.

-v : Searches for the non matching lines.

Using g++ to compile C++ files

G++ Compiler or GNU-C++ Compiler is a C++ compiler that follows the ISO C++ standard. Install it by running `sudo apt install g++`

We can compile C++ files by running g++ followed by the file name, by default an output file will be a.out which we will use to run our program.

Example: main.cpp

```
satharus@Argon:~$ cat main.cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "OSC" << endl;
    return 0;
}
satharus@Argon:~$ |
```

`g++ main.cpp`

```
satharus@Argon:~$ g++ main.cpp
satharus@Argon:~$ ls
a.out      Documents
Arduino    Downloads
```

```
satharus@Argon:~$ ./a.out
OSC
satharus@Argon:~$ |
```

Now the file a.out exists, it will be used to run our program. It is marked green as it is executable.

Test yourself:

Find an option for g++ to make you choose the name of the output file instead of compiling to the default file a.out.

Note: Byou can use `./file` to run an executable file from the Linux shell.

Solution:

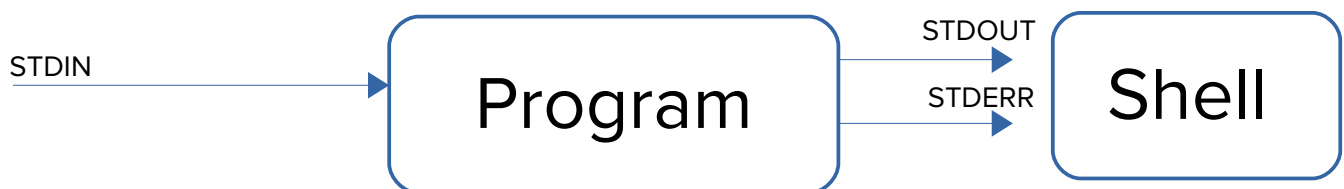
To change the name of the output file we can use the option `-o` followed by the output file name then the c++ file.

Example: `g++ -o output main.cpp`

Input and Output Streams and Redirection

Input and Output Streams

For every program, even operating systems, there is what we call standard input and output streams, they are used to take input from the user “standard input(stdin)”, and give the user output “standard output(stdout) and standard error(stderr)”.



These channels are the way to interact with any program, for example in Linux we can give the computer orders in the form of commands through the shell we call this stdin, an example for stdin will be `ls` or any other command, and the computer responds with either stdout or stderr depending on the input.

Running `ls` gives normal output, thus channeled to stdout:

```
satharus@Argon: ~/Arduino$ ls
libraries
satharus@Argon: ~/Arduino$ |
```

Running `hello` or any unknown command or syntax will give an error, thus channeled to stderr:

```
satharus@Argon: ~/Arduino$ hello
bash: hello: command not found
satharus@Argon: ~/Arduino$ |
```

Notice that in the previous examples, both the output and the error were printed to terminal as we haven't redirected either of them.


I/O Redirection

We can redirect stdout into file by using the greater than sign '>'.

Example:



```
satharus@Argon: ~/Arduino$ ls
libraries
satharus@Argon: ~/Arduino$ ls > output.txt
satharus@Argon: ~/Arduino$ cat output.txt
libraries
output.txt
satharus@Argon: ~/Arduino$ ls
libraries output.txt
satharus@Argon: ~/Arduino$ |
```



Notice that the output of the command was printed into the file instead of the terminal.



We can redirect stderr into file by using the sign '2>'.

Example:

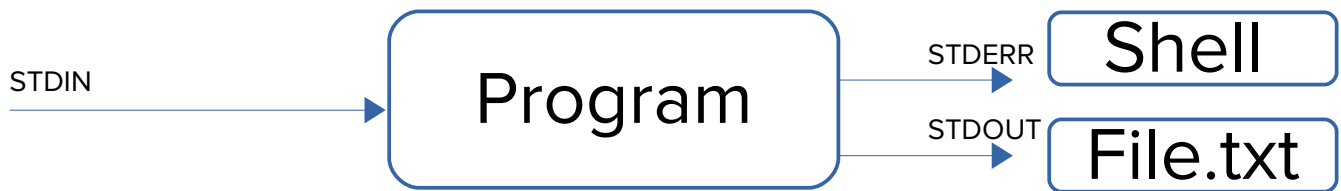
```
satharus@Argon: ~/Arduino$ hello
bash: hello: command not found
satharus@Argon: ~/Arduino$ hello > output2.txt
bash: hello: command not found
satharus@Argon: ~/Arduino$ cat output2.txt
satharus@Argon: ~/Arduino$ hello 2> err.txt
satharus@Argon: ~/Arduino$ cat err.txt
bash: hello: command not found
satharus@Argon: ~/Arduino$ |
```

Notice that the output here stayed on the terminal although we redirected stdout, that is because the type of the output is stderr. That's why when we used '2>' it redirected it into a file, but '>' didn't.

Using '>' will override what's in the file, to append to the file without deleting its content we can add another greater than sign '>>'.

You can also redirect input from a file into a program using the smaller than sign '<' followed by the filename. We can also use '&>' to redirect both.

Example: `./program < inputfile.txt`



Piping '|'

As we mentioned before, Linux allows us to manipulate those streams the way we want. One feature we can use is piping by using the or operator '|' or as we call it the pipe.

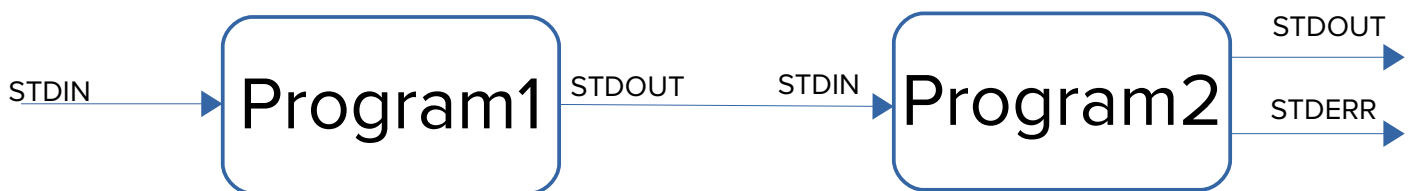
What the pipe does is that it takes any output passed to it then uses it as an input for the following command.

Example: `ls -l | grep "Desktop"`

```
satharus@Argon: ~$ ls -l | grep "Desktop"
drwxr-xr-x 7 satharus users 4096 Nov 29 18:04 Desktop
satharus@Argon: ~$ |
```

Executing this command will first list the directories in the current location and take this output rather than printing it, it passes it to the grep utility to filter out the lines that don't contain "Desktop".

We can use the pipe as many times as we want and to perform multiple commands sequentially with the I/O relations.



The figure above illustrates how piping works.

File Permissions

What permissions are

When we say permissions we usually refer to the type of access a certain user or group can have on a file.

The three permissions we have are:

Read: Which allows the user to view the content of a file.

Write: Which allows the user to edit on a file.

Execute: Which allows the user to run or execute a file.

To list permissions we use the command `ls` with the option `-l` which lists the files and directories in a list format.

```
satharus@Argon: ~/Public$ ls -l
total 8
drwxr-xr-x 2 satharus users 4096 Nov 29 18:48 dir1
drwxr-xr-x 2 satharus users 4096 Nov 29 18:48 dir2
-rw-r--r-- 1 satharus users  0 Nov 29 18:47 file1
-rw-r--r-- 1 satharus users  0 Nov 29 18:47 file2
lrwxrwxrwx 1 satharus users  5 Nov 29 18:48 link1 -> file1
satharus@Argon: ~/Public$ |
```

We can see at the beginning of every line there are a few sections:

```
drwxr-xr-x
drwxr-xr-x
-rw-r--r--
-rw-r--r--
lrwxrwxrwx
```

The first column is our focus

`[TYPE][OWNER PERMISSIONS][GROUP PERMISSIONS][OTHER PERMISSIONS]`

Type: This could be `d` for directory, `-` for file, `l` for link.



Permissions:

The three permissions are read 'r', write 'w', and execute 'x'.

The first three are the permissions for the owner of the file, the second three are for the group, the last three are for any users that are not in the two previously mentioned categories (others).

Example: Taking file1 as an example, The permissions are:

Owner: Read, Write.

Group: Read only.

Others: Read only.

And we know from the - at the start that the file is a regular file, not a directory or a link.

Changing Permissions

To change permissions we use the command `chmod`, and permissions are represented in 2 ways.

■ Absolute “Numeric” Method:

We can use the command `chmod` followed by a 3 digit number then the file we want to change its permissions.

Example: `chmod 764 file`

Each digit corresponds to a group of the three mentioned above and the value of the digit is the new permissions we’re giving the file, for each permission we have a number $x = 1$, $w = 2$, $r = 4$ so the command will set the owner’s permissions to read, write, execute (rwx) and the group to read and write (rw-) and others to read only (r--).

■ Symbolic Method:

Another way to change permissions is by referencing the group and the permissions.

u – Owner g – Group o – Others a – All users

Example: `chmod u=rw file`

This example sets the permissions of the owner to read and write only.

We can also use + and – to append or remove a certain permission.

Example: `chmod u+x file`

This example appends the execute permission to the owner of the file.

Changing Ownership

The owner of the file is usually the user who created it or anyone who was given ownership of the file.

The third and fourth columns of the output of the `ls -l` command show the user owner and the group owner of the file respectively.

```
satharus users  
satharus users  
satharus users  
satharus users  
satharus users
```

You can change the ownership of the file by using the command `chown` to change the user ownership and `chgrp` to change the group ownership.

Example: `chown lightdm dir2`

```
drwxr-xr-x 2 lightdm users 4096 Nov 29 18:48 dir2
```

Example: `chgrp wheel dir1`

```
drwxr-xr-x 2 satharus wheel 4096 Nov 29 18:48 dir1
```

Test yourself:

Let's have some fun, if you don't get something from the first time don't worry!

1. Nano, reading files, redirection, and compilation

- a) Create a file named file1.txt.
- b) Use the command `echo` to redirect the text "HELLO" into the file.
- c) Check that the file contains "HELLO" without using nano.
- d) Edit the file without using a GUI text editor to contain "GOODBYE".
- e) Find the line number that contains the word "GOODBYE" without using nano.
- f) Write a C++ program that does whatever you want.
- g) Compile the C++ program to produce an executable called "OSC".

2. Permissions and Ownership

- a) Create 2 files named filex and filey.
- b) Create 2 directories named dirx and diry.
- c) Change the permissions of filex to become NO PERMISSIONS for any of the users, using either of the methods.
- d) Change the permissions of filey to become (rw)(rx)(wx).
- e) Change the owner of dirx to root.
- f) Change the group of diry to root.

Solution:

1. Nano, reading files, redirection, and compilation

- a) `touch file1.txt`
- b) `echo "HELLO" > file1.txt`
- c) `cat file1.txt`
- d) `nano file1.txt`
- e) `cat -n file1.txt | grep "GOODBYE"`
- f) Just use Nano.
- g) `g++ filename -o OSC`

2. Permissions and Ownership

- a) `touch filex filey`
- b) `mkdir dirx diry`
- c) `chmod 000 filex`
`chmod a= filex`
`chmod u=,g=,o= filex`
- d) `chmod 653 filey`
`chmod u=rw,g=rx,o=wx filey`
- e) `sudo chown root dirx`
- f) `sudo chgrp root diry`