

# Lecture 3: Bash Language Part I

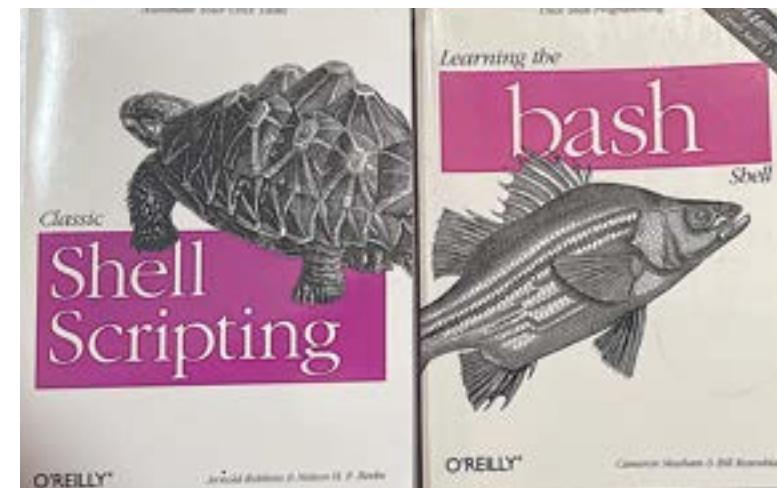


Photo of a new  
Linux user learning  
the command line



## Suggested Literature

Both books are available at  
<https://learning.oreilly.com>



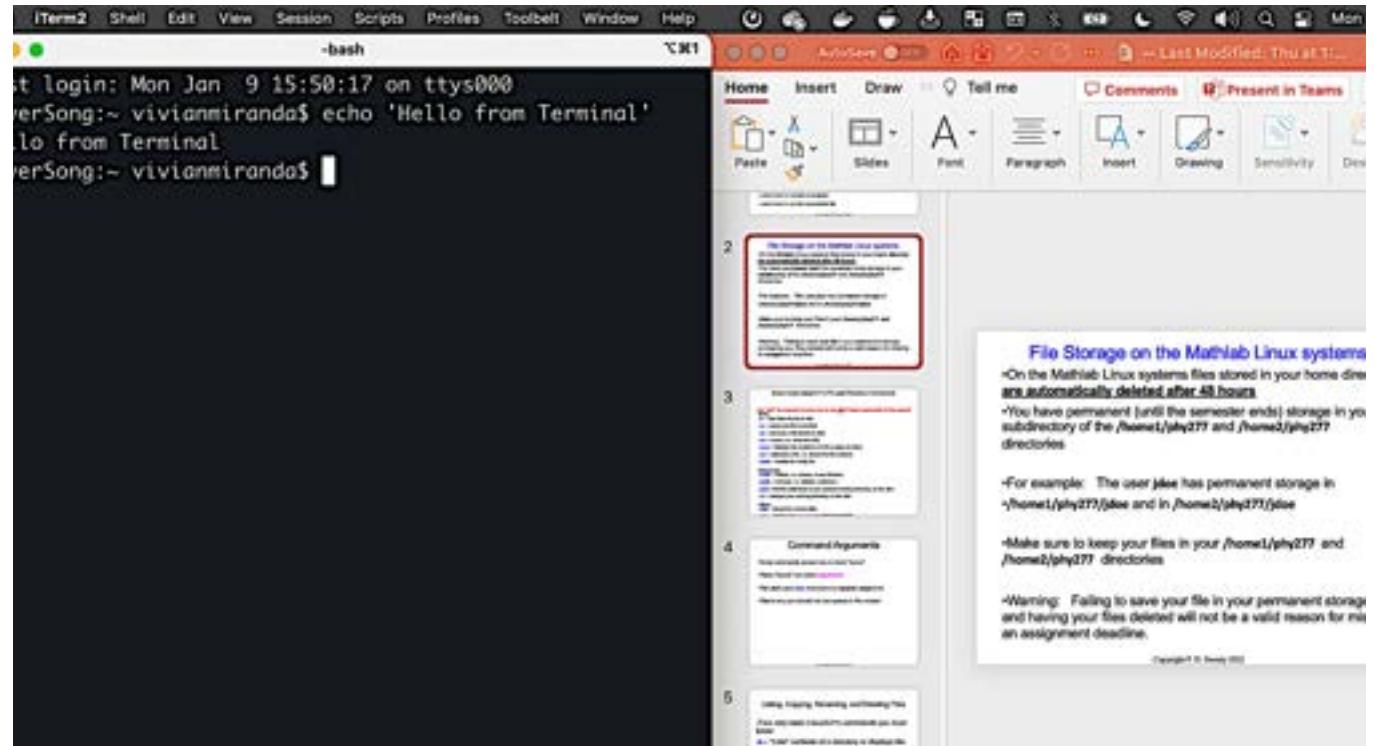
# GUI vs Terminal

Why using Terminal in 2023?

Cellphones have only GUI! Isn't Terminal too 1970's?

Key Factor is  
**automation**

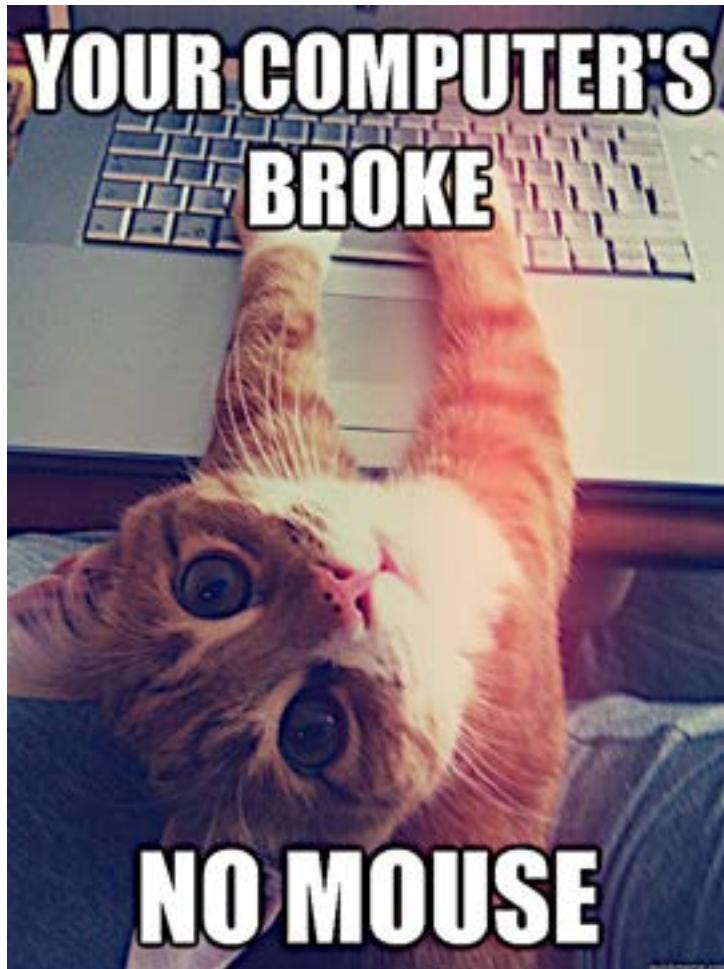
Ex: in research  
we sometimes  
need **scripts** to  
automate file  
handling



Bash is quite powerful. **OPEN THE TERMINAL**

# GUI vs Terminal

- You won't need to use `bash` all the time, but when you need it, the basic understanding of the following six lectures can save you so much time.



Most servers don't have VNC that allows remote GUI

Even with VNC, GUI requires a lot of bandwidth

We need to be comfortable with terminals

# What type of automation?

Dealing with strings is where bash shines



We will write a program that computes in < 1s

- How many unique words were written on Hamlet
- Show least-frequent words on Hamlet
- Show most-frequent words on Hamlet
- How many words were used just once?

# Piping: bash scripts is like playing Lego



We have many small pieces (programs).

From “gluing” of programs we build pipelines



```
RiverSong:Phy504Docker vivianmiranda$ docker run --platform linux/amd64  
--hostname phy504 -it -p 8080:8888 -v $(pwd):/home/whovian/host/ -v ~/  
.ssh:/home/whovian/.ssh:ro vivianmiranda/whovian-phy504
```

```
*****  
*  
* Welcome to Docker Whovian-PHY504 docker image. I hope this VM  
* will help you learn shell, C/C++/Fortran, and Python  
*  
* Homework for PHY504 must compile in the University machines, but I *  
* understand that running what we will learn on your laptop can help *  
* you set up your research software. You can also do your HW offline *  
* (ex: I take the train every day between Stony Brook and Queens), *  
* and then check if HW compiles on the lab machine afterward *  
*  
* This image also contains multiple gcc compilers (gcc 9 / 10 / 11) *  
* They contain new features about Fortran 2008 / C++17. Have fun! *  
*  
* PS: Files saved outside ~/host are deleted upon exiting the VM *  
*  
*****
```

You are running this container as user with ID 1000 and group 1000,  
which should map to the ID and group for your user on the Docker host.  
Great!

```
whovian@phy504:~$ █
```

Open the  
Linux  
terminal!

I will use  
docker to  
emulate  
Linux on  
MacOS

# Many shell commands are executable programs

```
whovian@phy504:/bin$ cd /bin  
whovian@phy504:/bin$ ls -l bash cat ls grep echo sort pwd head rm cp mv tr more  
-rwxr-xr-x 1 root root 1183448 Apr 18 2022 bash  
-rwxr-xr-x 1 root root 43416 Sep 5 2019 cat  
-rwxr-xr-x 1 root root 153976 Sep 5 2019 cp  
-rwxr-xr-x 1 root root 39256 Sep 5 2019 echo  
-rwxr-xr-x 1 root root 199136 Jan 29 2020 grep  
-rwxr-xr-x 1 root root 47480 Sep 5 2019 head  
-rwxr-xr-x 1 root root 142144 Sep 5 2019 ls  
-rwxr-xr-x 1 root root 43160 Feb 7 2022 more  
-rwxr-xr-x 1 root root 149888 Sep 5 2019 mv  
-rwxr-xr-x 1 root root 43352 Sep 5 2019 pwd  
-rwxr-xr-x 1 root root 72056 Sep 5 2019 rm  
-rwxr-xr-x 1 root root 117376 Sep 5 2019 sort  
-rwxr-xr-x 1 root root 51544 Sep 5 2019 tr
```

location of many system-wide programs



Another location of many executable program (including shell) is **/usr/bin** and **/usr/local/bin**

# Get the location of shell programs

```
whovian@phy504:/bin$ type -a echo
echo is a shell builtin
echo is /usr/bin/echo
echo is /bin/echo
whovian@phy504:/bin$ type -a more
more is /usr/bin/more
more is /bin/more
whovian@phy504:/bin$ type -a cp
cp is /usr/bin/cp
cp is /bin/cp
whovian@phy504:/bin$ type -a mv
mv is /usr/bin/mv
mv is /bin/mv
```

`type -a name` display all locations containing the executable name

Two locations may hold copies of the same program (in this case `/bin` is just a **symbolic link** of `/usr/bin`).

How does the system know where to look-up them (and the priority of execution)?

```
whovian@phy504:/bin$ echo "$PATH"
/opt/conda/bin:/usr/local/nvidia/bin:/usr/local/cuda/bin:/usr/local/sbin
:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

# Shell programs are written in C

```
int
main (int argc, char **argv)
{
    int c;
    bool ok;
    bool make_backups = false;
    char *backup_suffix_string;
    char *version_control_string = NULL;
    struct cp_options x;
    bool copy_contents = false;
    char *target_directory = NULL;
    bool no_target_directory = false;

    initialize_main (&argc, &argv);
    set_program_name (argv[0]);
    setlocale (LC_ALL, "");
    bindtextdomain (PACKAGE, LOCALEDIR);
    textdomain (PACKAGE);

    atexit (close_stdin);

    selinux_enabled = (0 < is_selinux_enabled ());
    cp_option_init (&x);

    /* FIXME: consider not calling getenv for SIMPLE_BACKUP_SUFFIX unless
       we'll actually use backup_suffix_string.  */
    backup_suffix_string = getenv ("SIMPLE_BACKUP_SUFFIX");

    while ((c = getopt_long (argc, argv, "abdfHillNprst:uvxPRS:T",
                           long_opts, NULL))
           != -1)
```

wertarbyte / **coreutils** Public

<> Code Issues 3 Pull requests 1 Actions

master coreutils / src / cp.c

pixelb cp: add an option to only copy the file attributes ...

C language is the base  
of Linux kernel and  
shell scripts

**cp** code is >1000 lines!  
Shell can do so much  
with just a few lines

# EDITORS – EMACS NANO and VIM

- Popular editors on Unix: **Emacs & vim & nano** (all installed on Docker)
- On a command line interface all editor functions are available as keyboard commands
- Makes it possible to edit when remotely logged in
- I use **nano** – simpler for this course. (**my suggestion**)

VIM

usable in just about  
any environment.

does one thing, well.



EMACS

flexible, customizable, and  
packed with every feature  
known to man.



NANO

mostly used by people  
who do not know  
what they are doing;  
or psychopaths.



# Open GNU nano

The screenshot shows a terminal window titled "com.docker.cli" with the title bar "GNU nano 4.8" and status indicators "New Buffer" and "Modified". The window contains the following text:

```
Hi, this is GNU nano editor
There are some instructions below on how to manipulate this file.
If you want to exit the file, then press CRTL+X.
Then decide if you want to save the file or not!

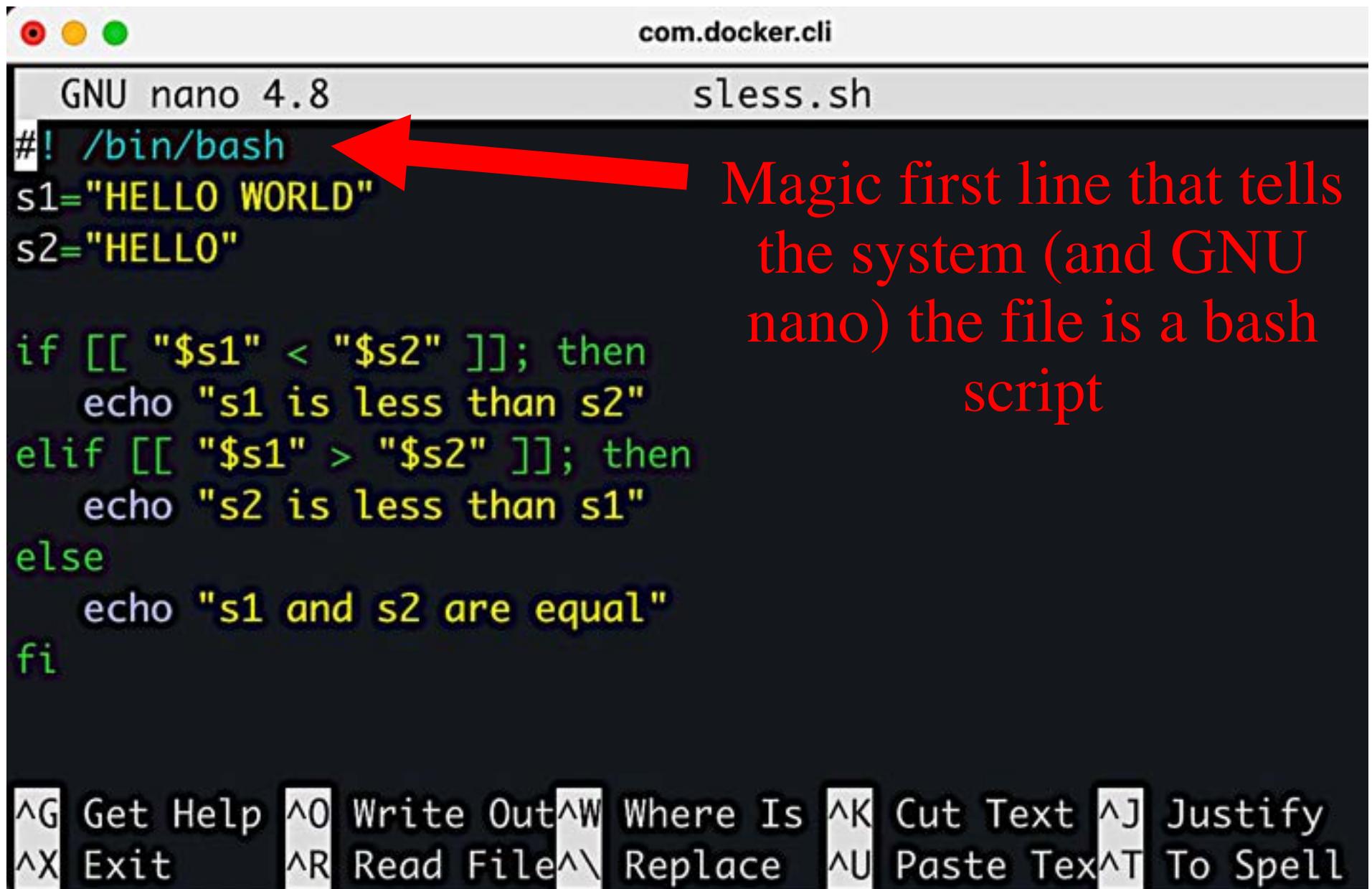
Nano is not just black and white - there is a special line you
can add so that nano knows how to place colors (depending on
the language)
```

At the bottom, a series of keyboard shortcuts are listed:

$\wedge G$	Get Help	$\wedge O$	Write Out	$\wedge W$	Where Is	$\wedge K$	Cut Text	$\wedge J$	Justify
$\wedge X$	Exit	$\wedge R$	Read File	$\wedge \backslash$	Replace	$\wedge U$	Paste Text	$\wedge T$	To Spell

HW: Practice working with GNU nano

# GNU nano is not just B&W



The screenshot shows the GNU nano 4.8 text editor interface. The title bar indicates the file is named "sless.sh". The editor window displays the following content:

```
GNU nano 4.8
#!/bin/bash
s1="HELLO WORLD"
s2="HELLO"

if [[ "$s1" < "$s2" ]]; then
    echo "s1 is less than s2"
elif [[ "$s1" > "$s2" ]]; then
    echo "s2 is less than s1"
else
    echo "s1 and s2 are equal"
fi
```

A red arrow points to the line `#!/bin/bash`, which is highlighted in blue. To the right of the arrow, the text "Magic first line that tells the system (and GNU nano) the file is a bash script" is displayed in red.

At the bottom of the screen, a menu bar is visible with the following options:

- ^G Get Help
- ^O Write Out
- ^W Where Is
- ^K Cut Text
- ^J Justify
- ^X Exit
- ^R Read File
- ^V Replace
- ^U Paste Tex
- ^T To Spell

# How to limit page size in nano?

Add a mark with `-J num` option (does not limit page size, just a mark)

Options is core feature that makes bash/terminal a powerful tool

```
whovian@phy504:~/host$ nano -J 40 sless.sh
```

```
GNU nano 4.8 com.docker.cli
#!/bin/bash
s1="HELLO WORLD"
s2="HELLO"

if [[ "$s1" < "$s2" ]]; then
    echo "s1 is less than s2"
elif [[ "$s1" > "$s2" ]]; then
    echo "s2 is less than s1"
else
    echo "s1 and s2 are equal"
fi

^G Get Help ^O Write Out ^W Where Is ^K Cut Text
^X Exit      ^R Read File ^\ Replace ^U Paste Te
```

In your research,  
you can avoid using  
GNU nano with `sftp`  
connection to the  
server.

But in some places  
like SBU, there is  
nightmarish 2-factor  
authentication on  
sftp

# How to get help in terminal commands?

The man command displays the manual page for the command

Many times, the option **--help** display a succinct manual version

```
LS(1)                                     General Commands Manual

NAME
    ls - list directory contents

SYNOPSIS
    ls [-@ABCDEFGHIOPRSTUWabcdefghijklmnopqrstuvwxyz1%,] [--color=when] [-D format] [file ...]

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

    If no operands are given, the contents of the current dire
    than one operand is given, non-directory operands are disp
    directory operands are sorted separately and in lexicograph

    The following options are available:
```



PS: docker phy504 container does not have **man** installed. Use **--help**

```
whovian@phy504:~/host$ bash --help
GNU bash, version 5.0.17(1)-release-(x86_64-pc-linux-gnu)
Usage:  bash [GNU long option] [option] ...
        bash [GNU long option] [option] script-file ...
GNU long options:
  --debug
  --debugger
  --dump-po-strings
  --dump-strings
  --help
  --init-file
  --login
  --noediting
  --noprofile
  --norc
  --posix
  --pretty-print
  --rcfile
  --restricted
  --verbose
  --version
Shell options:
  -ilrsD or -c command or -O shopt_option          (invocation only)
  -abefhkmnptuvxBCHP or -o option
Type `bash -c "help set"' for more information about shell options.
Type `bash -c help' for more information about shell builtin commands.
Use the `bashbug' command to report bugs.

bash home page: <http://www.gnu.org/software/bash>
General help using GNU software: <http://www.gnu.org/gethelp/>
```

Even the word  
bash (which is a  
program) accept  
--help

# Basic File Manipulation

# Creating New Directories

New directories can be created with the **mkdir** (make directory)

Syntax:

**mkdir <name\_of\_new\_directory>**

Example:

**mkdir codes**

Creates a new directory named **codes**

```
whovian@phy504:~$ mkdir --help
Usage: mkdir [OPTION]... DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too
 -m, --mode=MODE      set file mode (as in chmod), not a=rwx - umask
 -p, --parents        no error if existing, make parent directories as
 -v, --verbose        print a message for each created directory
 -Z                  set SELinux security context of each created directory
                     to the default type
 --context[=CTX]      like -Z, or if CTX is specified then set the SELinux
                     or SMACK security context to CTX
 --help               display this help and exit
```

# The **ls** command

**ls** displays all files in the current directory (with information)

- **-d** list directories not files (**options can tweak command behavior**)
- **-S** (yes CAPS): sort files by size, largest first
- **-t** sort by modification time, newest first
- **-U** (yes CAPS) do not sort
- **-l** use long listing format (with size in bytes)
- **-h** (human readable)! That means converting bytes to Kb, Mb, Gb

```
whovian@phy504:~/host$ ls -Shl      Yes! Concatenate options
total 428K
-rw-r--r-- 1 whovian users 175K Jan 13 00:22 hamlet.txt
-rw-r--r-- 1 whovian users 124K Jan 12 02:35 es.txt
-rw-r--r-- 1 whovian users  556 Jan 12 21:21 compile_des_y3
-rw-r--r-- 1 whovian users   416 Jan 13 00:38 wf.sh
-rw-r--r-- 1 whovian users   391 Jan 17 16:45 employees.txt
-rw-r--r-- 1 whovian users   342 Jan 20 01:22 address_list.txt
-rw-r--r-- 1 whovian users   341 Jan 20 01:06 address_list_backup.txt
-rw-r--r-- 1 whovian users   326 Jan 17 04:02 caseloop.sh
-rw-r--r-- 1 whovian users   283 Jan 17 03:31 whileloop3.sh
```

# The **ls** command

**ls** displays all files in the current directory (with information)

- **-d** list directories not files (**options can tweak command behavior**)
- **-S** (yes CAPS): sort files by size, largest first
- **-t** sort by modification time, newest first
- **-U** (yes CAPS) do not sort
- **-l** use long listing format (with size in bytes)
- **-h** (human readable)! That means converting bytes to Kb, Mb, Gb

```
whovian@phy504:~/host$ ls -S  without -l = short
hamlet.txt          caseloop.sh      whileloop.sh    myfunc2.sh
es.txt              whileloop3.sh   hello.txt       example
compile_des_y3     whileloop2.sh   ifelse3.sh     example2
wf.sh               sless.sh        forloop.sh     ifelse2.sh
employees.txt       table.txt       ifelse.sh      portugues-numbers
address_list.txt    sorting2.txt    singers.txt   forloop2.sh
address_list_backup.txt  singers2.txt  sorting3.txt  wine.txt
```

# The **ls** command

One Useful acronym is **latr** (long, all, by time and reverse)

```
whovian@phy504:~/host/C$ ls -latr | tail -n 10
-rw-r--r--  1 whovian users    436 Jan 26 18:49 functions8.c
-rw-r--r--  1 whovian users    274 Jan 26 22:15 functions9.c
-rwxr-xr-x  1 whovian users 16760 Jan 26 22:15 functions9
-rw-r--r--  1 whovian users    278 Jan 26 22:43 functions10.c
-rwxr-xr-x  1 whovian users 16728 Jan 26 22:43 a.out
-rwxr-xr-x  1 whovian users 16728 Jan 26 22:45 functions10
-rw-r--r--  1 whovian users    282 Jan 26 22:54 functions11.c
-rwxr-xr-x  1 whovian users 16824 Jan 26 22:54 functions11
drwxr-xr-x 86 whovian users  2752 Jan 27 02:18 .
drwxr-xr-x 53 whovian users  1696 Jan 27 13:34 ..
```

Why it is Useful? Sometimes, you need to check permissions  
of files you just work on

# Hidden Files

**Some files / folders are hidden (not shown by default `ls` and GUI)**

```
whovian@phy504:~$ ls  
bash.bashrc host stablesort.txt  
whovian@phy504:~$ ls -d .*  
. .. .cache .cmake .conda .jupyter .local .ssh  
whovian@phy504:~$ more .conda/  
  
*** .conda/: directory ***
```

They are usually cache / system files  
(files that will crash the OS if deleted)

Hidden files on Unix start with .

# Hidden Files

Some files / folders are hidden (not shown by default `ls` and GUI)

```
whovian@phy504:~/host/teste_hidden$ ls -Shl
total 12K
-rw-r--r-- 1 whovian users 283 Jan 20 02:41 whileloop3.sh
-rw-r--r-- 1 whovian users 239 Jan 20 02:41 whileloop2.sh
-rw-r--r-- 1 whovian users 146 Jan 20 02:41 whileloop.sh
whovian@phy504:~/host/teste_hidden$ ls -aShl
total 16K
drwxr-xr-x 44 whovian users 1.4K Jan 20 02:41 ..
-rw-r--r-- 1 whovian users 283 Jan 20 02:41 whileloop3.sh
-rw-r--r-- 1 whovian users 239 Jan 20 02:41 whileloop2.sh
drwxr-xr-x 6 whovian users 192 Jan 20 02:41 .
-rw-r--r-- 1 whovian users 146 Jan 20 02:41 whileloop.sh
-rw-r--r-- 1 whovian users 14 Jan 20 02:41 .myhiddenfile.txt
```

These are file permissions (you may not be allowed to write on (or execute) all files

# The **cp** (copy) and **mv** (move) commands

**cp** or **mv** commands accept new filename or directories

as their second args. If dir then type either relative or absolute paths

```
whovian@phy504:~/host$ ls  
es.txt  
whovian@phy504:~/host$ cp es.txt es2.txt  
whovian@phy504:~/host$ mkdir test  
whovian@phy504:~/host$ cp es.txt ./test/  
whovian@phy504:~/host$ ls  
es2.txt es.txt test  
whovian@phy504:~/host$ ls ./test/  
es.txt  
whovian@phy504:~/host$ ls  
es2.txt es.txt test  
whovian@phy504:~/host$ mkdir test2  
whovian@phy504:~/host$ mv es2.txt test2/  
whovian@phy504:~/host$ ls  
es.txt test test2  
whovian@phy504:~/host$ ls ./test2/  
es2.txt  
whovian@phy504:~/host$ █
```

**cp** or **mv** commands  
have many options. Try

**cp --help**

and

**mv --help**

**Copy directories**

**cp -r**

one dash ‘-’ and **r**

**HW: research one vs  
two dashes**

# Parameter expansion

If you need to create many directories, use parameter expansion

```
whovian@phy504:~/host/test$ mkdir data{0..12}
whovian@phy504:~/host/test$ ls
data0  data1  data10  data11  data12  data2  data3  data4  data5  data6  data7  data8  data9
```

If you need to move / copy many files, use parameter expansion

```
whovian@phy504:~/host/test$ ls
data0  data1  data10  data11  data12  data2  data3  data4  data5  data6  data7  data8  data9  tmp2
whovian@phy504:~/host/test$ cp -r data{0..12} tmp2/
whovian@phy504:~/host/test$ ls ./tmp2/
data0  data1  data10  data11  data12  data2  data3  data4  data5  data6  data7  data8  data9
```

Parameter expansion also works with different file types

```
whovian@phy504:~/host/test$ touch file.txt file.png
whovian@phy504:~/host/test$ cp file{.txt,.png} ./tmp2/
whovian@phy504:~/host/test$ ls tmp2/
data0  data1  data10  data11  data12  data2  data3  data4  data5  data6  data7  data8  data9  file.png  file.txt
whovian@phy504:~/host/test$ █
```

# Lecture 4: Bash Language Part II

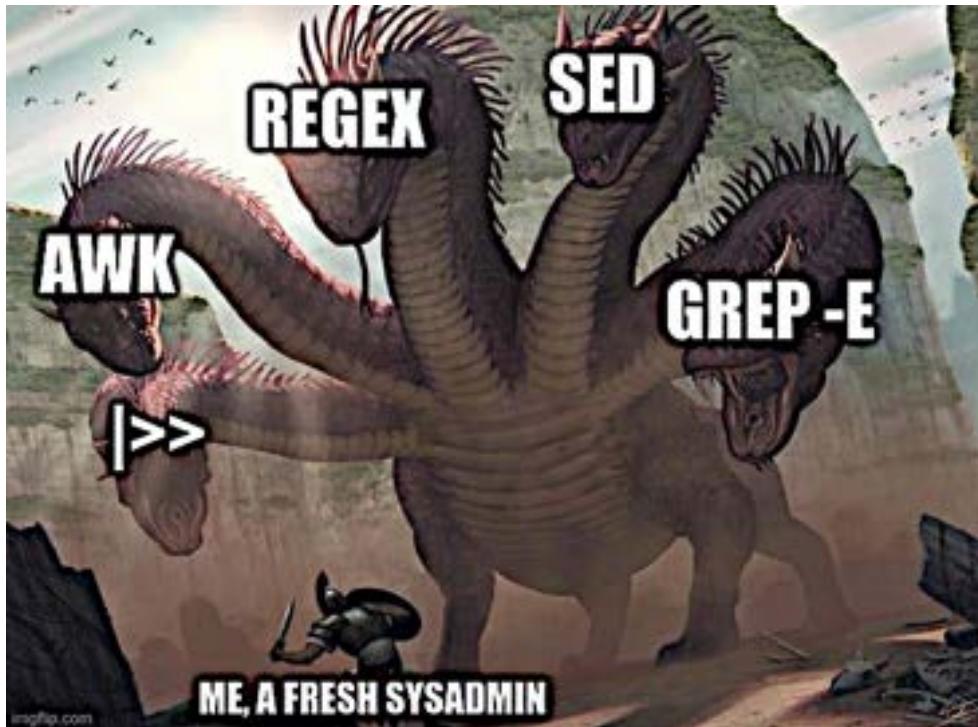
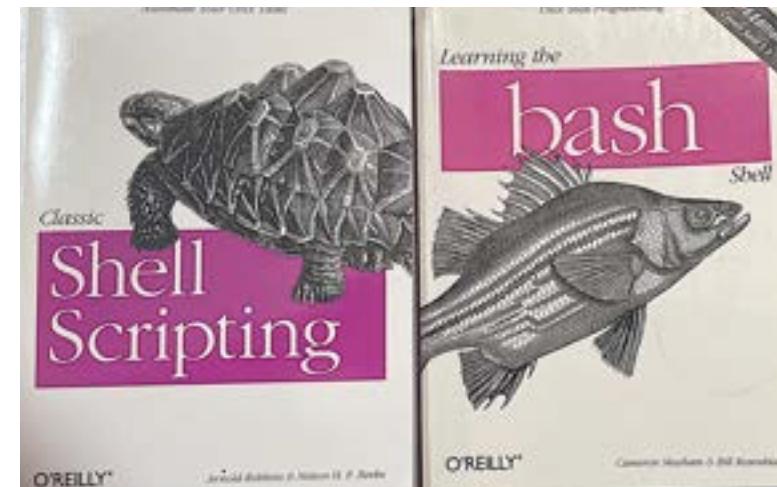


Photo of a new  
Linux user learning  
the command line



## Suggested Literature

Both books are available at  
<https://learning.oreilly.com>



# Basic File Manipulation (continuation)

# Absolute Pathnames

- Absolute paths can be used to change working directories. Try this:

`cd /`

This command changes your working directory to the root dir

Verify this with by executing the `pwd` command!

Return to your home directory? `cd` command with no arguments

```
whovian@phy504:~$ pwd  
/home/whovian  
whovian@phy504:~$ cd /  
whovian@phy504:/$ pwd  
/  
whovian@phy504:/$ ls  
bin dev home lib32 libx32 mnt opt root sbin sys usr  
boot etc lib lib64 media NGC-DL-CONTAINER-LICENSE proc run srv tmp var  
whovian@phy504:/$ cd  
whovian@phy504:~$ pwd  
/home/whovian
```

**pwd provides the absolute path of the current directory**

**ls has many interesting options**

```
whovian@phy504:~$ ls --help  
Usage: ls [OPTION]... [FILE]...  
List information about the FILEs (the current directory by default).  
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
```

**check ls --help**

# Relative Pathnames

Sometimes absolute paths are the quickest way to refer to a file/dir

Absolute path to a file or directory is that it may be long

```
RiverSong:Vivian vivianmiranda$ ls  
Docker          PHY504.pptx  
RiverSong:Vivian vivianmiranda$ pwd  
/Users/vivianmiranda/Library/CloudStorage/OneDrive-StonyBrookUniversity/teaching/Phy504/Vivian
```

**My laptop**

This could make commands unwieldy! Is there an easier way?

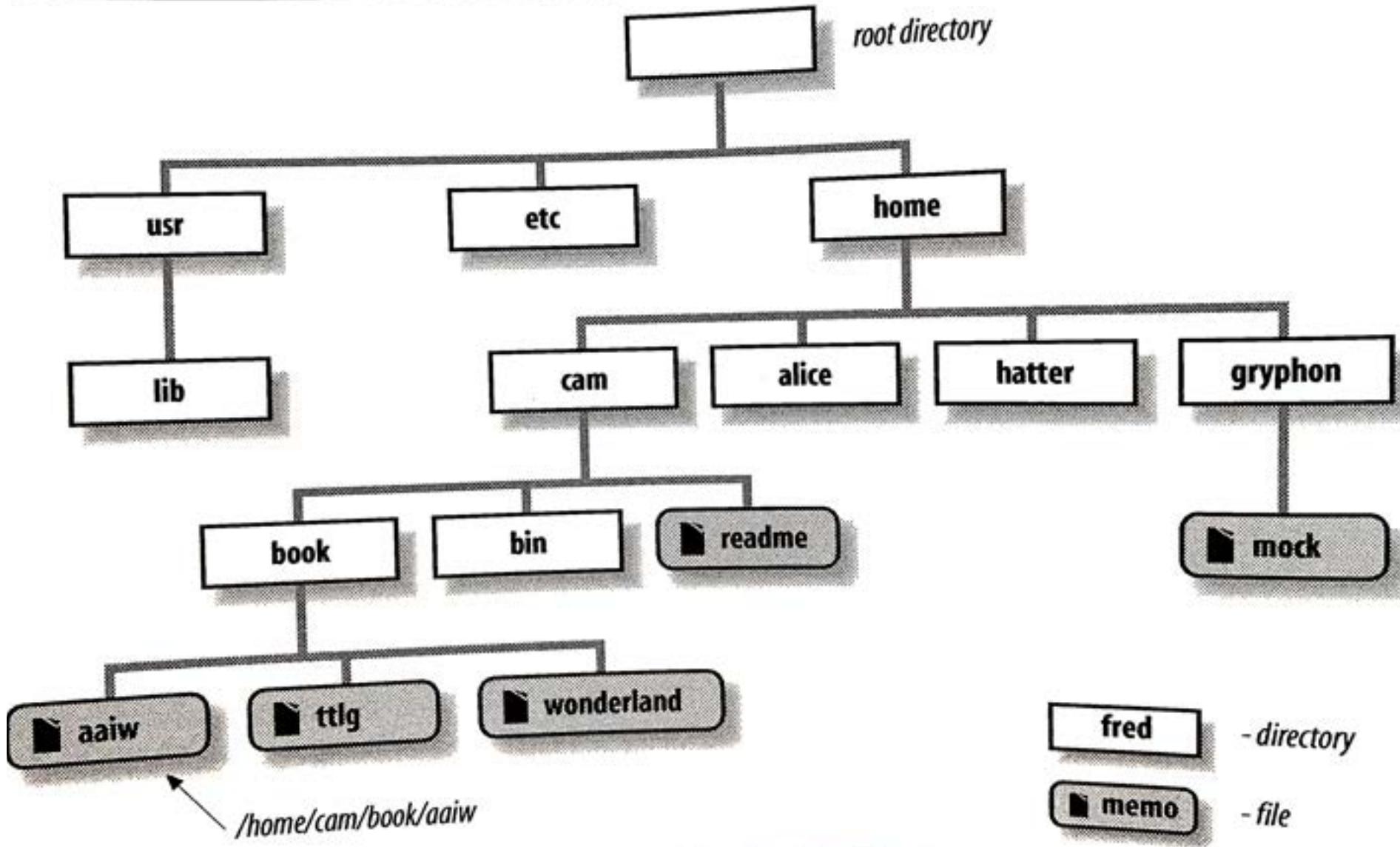
A **relative path** or **relative pathname** specifies the position of a file/dir relative to some other entity the shell knows about

```
RiverSong:Vivian vivianmiranda$ ls ./Docker/   from current dir  
RiverSong:Vivian vivianmiranda$ ls ~/  
Applications           from home dir  OneDrive - Stony Brook University  
Applications (Parallels)          Parallels
```

```
RiverSong:Vivian vivianmiranda$ ls ../      from parent directory  
PHY277          PHYS305_2022     Shared          Vivian          bash  
RiverSong:Vivian vivianmiranda$ ls ../../..  from grandparent  
Chicago DL      Phy504       SM.pdf    books  
RiverSong:Vivian vivianmiranda$ █          directory
```

# ? Parent / Grandparent directories ?

Directories in Unix system form a directory tree



# Deleting Directories

Directories can be removed (deleted) with the **rmdir** (Remove Directory)  
**IF THEY ARE EMPTY – THERE IS NO TRASH – BE CAREFUL**

Syntax:

**rmdir <name\_of\_new\_directory>**

Example:

**rmdir codes**

Deletes the directory named **codes**

```
whovian@phy504:~$ rmdir --help
Usage: rmdir [OPTION]... DIRECTORY...
Remove the DIRECTORY(ies), if they are empty.

--ignore-fail-on-non-empty
          ignore each failure that is solely because a directory
          is non-empty
-p, --parents    remove DIRECTORY and its ancestors; e.g., 'rmdir -p a/b'
                  similar to 'rmdir a/b/c a/b a'
-v, --verbose   output a diagnostic for every directory processed
--help        display this help and exit
--version   output version information and exit
```

# Deleting Files/Dir

Directories can be removed (deleted) with the **rm** (Remove file)

**BE CAREFUL WITH rm**

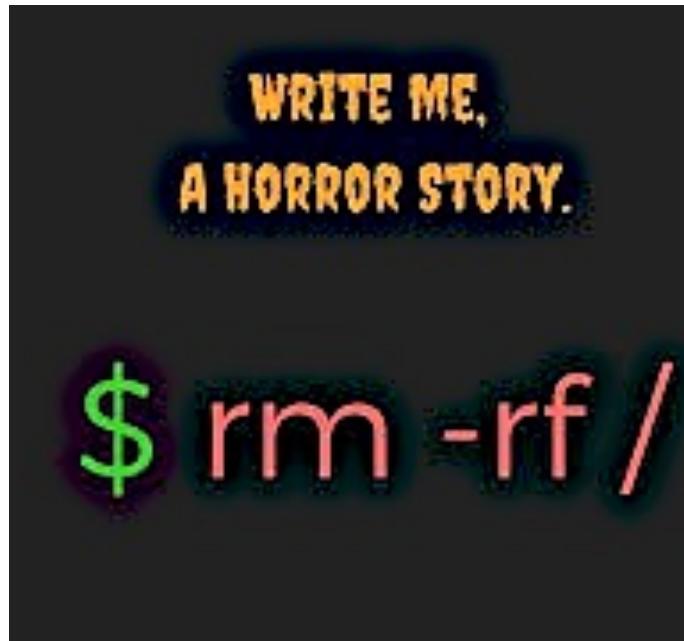
Syntax:

**rm <name\_of\_new\_file>**

Example:

**rm codes.txt**

Deletes the file named **codes**



**BE CAREFUL WITH rm**

This command deletes  
**ALL** your files/dir

-r (dir), -f (force)

**Power comes with responsibility**

**You can erase all files in your laptop with wrong command**

**Terminal with docker**

Don't give them access to all your files. Create an empty local folder

Example: I created `/Users/vivianmiranda/data/teaching/Phy504Docker`

Power comes with responsibility  
**You can erase all files in your laptop with wrong command**

## Terminal with docker

Your files will be seen inside the VM at **/home/whovian/host/**  
Everything in the VM outside **~/host/** is **ERASED** when you exit

```
RiverSong:Phy504Docker vivianmiranda$ ls
2 es.txt example.sh hello.txt io.txt singers2.txt
singers.txt
```

```
whovian@phy504:~$ cd ~/host/
whovian@phy504:~/host$ ls
2 es.txt example.sh hello.txt io.txt singers2.txt singers.txt
```

**Warning:** Failing to save your file in your permanent storage and having your files deleted will not be a valid reason for missing an assignment deadline.

# The **more** command

The **more** command displays a text file (quite powerful features)

```
RiverSong:lects vivianmiranda$ more -s l21_equations.tex  
\documentclass[12pt]{article}  
\usepackage[left=1.0in,right=1.0in]{geometry}  
\geometry{letterpaper,landscape,margin=0.75in}
```

```
\begin{document}
```

**-s squeeze multiple  
blank lines into one**

```
Slide 3 equation 1:
```

The **-s** option squeezes multiple blank lines into one single blank line

```
RiverSong:lects vivianmiranda$ more +35 l21_equations.tex  
\[  
f(x) = x^2  
\]  
with $x_L=1$ and $x_H=2$.  
press 'q' to quit  
displaying the file
```

The **+num** skips the first num lines

Powerful feature **+/pattern** find a pattern  
(word or **regular expression**)

# The `head` command

The `head` command displays the beginning of a text file

```
whovian@phy504:~/host$ head -n 2 es.txt
Aadil
Aali
whovian@phy504:~/host$ head -n 4 es.txt
Aadil
Aali
Aaliyah
Aaltje
whovian@phy504:~/host$ head -n 2 -v es.txt
==> es.txt <==
Aadil
Aali
```

default

behavior: show

first 10 lines

- The `-v` option = verbose (show file name)
- The `-n` option = number of lines to show
- Interesting fact about `-n` option

# The `head` command

- The `-n` option also accepts negative integers!

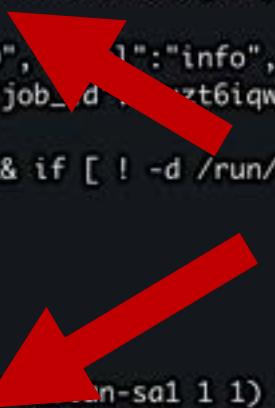
```
whovian@phy504:~/host$ more singers.txt
4 Avril_Lavigne.
10 Rihanna.
2 Lady_Gaga.
5 Usher
14 Shakira
1 Justin_Timberlake.
20 Billie_Joe_Armstrong
3 Taylor_Swift.
whovian@phy504:~/host$ head -n -2 singers.txt
4 Avril_Lavigne.
10 Rihanna.
2 Lady_Gaga.
5 Usher
14 Shakira
1 Justin_Timberlake.
```

Negative number (example `-2`) = print ALL but the last two

# The tail command

Finding in real time who is trying to **hack** my server via VNC

```
vivianmiranda@amypond:/var/log$ tail -f -n 10 /var/log/syslog
Jan 26 14:37:02 amypond vncserver-virtuald[1610]: Connections: connected: 94.232.45.214::61000 (TCP)
Jan 26 14:37:34 amypond vncserver-virtuald[1610]: Connections: disconnected: 94.232.45.214::61000 (TCP) ([System-10
4] read: Connection reset by peer (104))
Jan 26 14:38:00 amypond mattermost[6826]: {"timestamp":"2023-01-26 14:38:00.782 -05:00", "level": "info", "msg": "Simpl
eWorker: Job is complete", "caller": "jobs/base_workers.go:88", "worker": "ExpiryNotify", "job_id": "zxt6iqwp3ybdcnx687n
3puffh"}
Jan 26 14:39:01 amypond CRON[362266]: (root) CMD ( [ -x /usr/lib/php/sessionclean ] && if [ ! -d /run/systemd/syst
em ]; then /usr/lib/php/sessionclean; fi)
Jan 26 14:39:06 amypond systemd[1]: Starting Clean php session files...
Jan 26 14:39:06 amypond systemd[1]: phpsessionclean.service: Succeeded.
Jan 26 14:39:06 amypond systemd[1]: Finished Clean php session files.
Jan 26 14:42:07 amypond systemd[1]: Started Session 309 of user vivianmiranda.
Jan 26 14:45:01 amypond CRON[363709]: (root) CMD (command -v debian-sa1 > /dev/null && /usr/lib/cgi-bin/sa1 1 1)
Jan 26 14:45:18 amypond vncserver-x11[1664,root]: Connections: disconnected: 92.255.85.183::64623 (TCP) ([IdleTimeo
ut] The authentication period has timed out.)
```



IP address details

**94.232.45.214**

 Sofia, Sofia-Capital, Bulgaria

IP address details

**92.255.85.183**

 Moscow, Moscow, Russia

# The **cut** command

**cut** allows you to cut out sections of a specified file or piped data and print the result to standard output

```
whovian@phy504:~/host$ more employees.txt
Adeline Bird      clerk      $654
Weston Bradley   clerk      $789
Anaya Rice       director   $1334
Andre Wilkins    clerk      $446
Ian Norris       hygiene    $655
Omar Landry     accountant $465
Harley Edwards   secretary  $4563
Marie Snow        manager   $4566
Haylee Sweeney   CEO       $12334
Isis Ware        manager   $4566
Jaelyn Khan     researcher $789
```

To extract the first char (**-c 1**) from each input line, run:

```
whovian@phy504:~/host$ cut -c 1 employees.txt
A
W
A
A
I
O
H
M
H
I
J
```

```
whovian@phy504:~/host$ cut -c 1-5 employees.txt
Adeli
Westo
Anaya
Andre
Ian N
Omar
Harle
Marie
```

You can also extract char ranges with **-c X-Y**

You can also extract multiple ranges with **-c X-Y, Z-M**

# The **cut** command x2 (based on delimiters)

**cut** allows you to cut out sections of a specified file or piped data and print the result to standard output.

To extract based on delimiters, e.g., spaces, use (**-d**)

```
whovian@phy504:~/host$ echo "Miranda is an IT provider" | cut -d ' ' -f 1
Miranda
whovian@phy504:~/host$ echo "Miranda is an IT provider" | cut -d ' ' -f 2
is
whovian@phy504:~/host$ echo "Miranda is an IT provider" | cut -d ' ' -f 3
an
whovian@phy504:~/host$ echo "Miranda is an IT provider" | cut -d ' ' -f 1-
Miranda is an IT provider
whovian@phy504:~/host$ echo "Miranda is an IT provider" | cut -d ' ' -f 1-3
Miranda is an
```

PS: the default delimiter is the TAB character

# The **cut** command x3 (output delimiters)

**cut** allows you change the delimiter on output w/ **--output-delimiter**

```
whovian@phy504:~/host/C$ grep whovian /etc/passwd
whovian:x:1000:1000::/home/whovian:/bin/bash
whovian@phy504:~/host/C$ grep whovian /etc/passwd | cut -d ':' -f 1,7
whovian:/bin/bash
whovian@phy504:~/host/C$ grep whovian /etc/passwd | cut -d ':' -f 1,7 --output-delimiter=$'\n'
whovian
/bin/bash
```

Another cool option is **-s**, cut only cuts when there is the delimiter

```
whovian@phy504:~/host/C$ echo "HelloWorld" | cut -d " " -f 1 -s
whovian@phy504:~/host/C$ echo "Hello World" | cut -d " " -f 1 -s
Hello
```

Finally, there is **--complement** (self-explanatory)

```
whovian@phy504:~/host/C$ echo "Hello to the whole World" | cut -d " " -f 1
Hello
whovian@phy504:~/host/C$ echo "Hello to the whole World" | cut -d " " -f 1 --complement
to the whole World
```

# The **join** command

For pair of input lines with identical fields, write a line to output

```
whovian@phy504:~/host$ more foodtypes.txt
1 Protein
2 Carbohydrate
3 Fat
whovian@phy504:~/host$ more foods.txt
1 Cheese
2 Potato
3 Butter
whovian@phy504:~/host$ join foodtypes.txt foods.txt
1 Protein Cheese
2 Carbohydrate Potato
3 Fat Butter
```

```
whovian@phy504:~/host$ more foods2.txt
2 Potato
1 Cheese
3 Butter
whovian@phy504:~/host$ join foodtypes.txt foods2.txt
join: foods2.txt:2: is not sorted: 1 Cheese
2 Carbohydrate Potato
3 Fat Butter
```

Attention: field  
**\$1** needs to be  
sorted!

# The `join` command x2

For pair of input lines with identical fields, write a line to output

What if the identical field is not column 1?

In that case use the command options

`-1 COL_FILE_1 -2 COL_FILE_2`

```
whovian@phy504:~/host$ more wine.txt
White Reisling Germany
Red Riocha Spain
Red Beaunes France
whovian@phy504:~/host$ more reviews.txt
Reisling Terrible!
Riocha Meh
Beaunes Great!
whovian@phy504:~/host$ join -1 2 -2 1 wine.txt reviews.txt
Reisling White Germany Terrible!
Riocha Red Spain Meh
Beaunes Red France Great!
```



# The echo command

The **echo** command displays line of text/string passed as an argument

```
whovian@phy504:~/host$ echo 'Hello 504 Class. How are you?'
Hello 504 Class. How are you?
whovian@phy504:~/host$ echo -e '\nHello 504 Class. How are you?\n'

Hello 504 Class. How are you?

whovian@phy504:~/host$ echo -e 'Hello 504 Class. \tHow are you?'
Hello 504 Class.           How are you?
whovian@phy504:~/host$ echo -e 'Hello 504 Class. \bHow are you?'
Hello 504 Class.How are you?
whovian@phy504:~/host$ echo -e 'Hello 504 Class. \vHow are you?'
Hello 504 Class.

How are you?
```

- The **-e** option allows the backlash to be interpreted
- The **-n** option does not skip line

Sequence	Description
\a	Alert character, usually the ASCII BEL character.
\b	Backspace.
\c	Suppress the final newline in the output. Furthermore, any characters left in the argument, and any following arguments, are ignored (not printed).
\f	Formfeed.
\n	Newline.
\r	Carriage return.
\t	Horizontal tab.
\v	Vertical tab.

# The `echo` command part 2

`echo` displays line of text/string passed as an argument

What if we want to redirect the output of echo to a file?

```
whovian@phy504:~/host$ echo "Hello world" >> hello.txt
whovian@phy504:~/host$ more hello.txt
Hello world
```

The `echo` outputs variables. What if we want to output the content of files. Use the command `cat`  
(copy input – the file – to the output – the terminal)

```
whovian@phy504:~/host$ echo hello.txt
hello.txt
whovian@phy504:~/host$ cat hello.txt
Hello world
```

## tac & rev commands

tac displays line of a file in reverse ordering

rev reverses chars on each string (but not the line ordering)

```
whovian@phy504:~/host$ cat wine.txt
White Reisling Germany
Red Riocha Spain
Red Beaunes France
whovian@phy504:~/host$ tac wine.txt
Red Beaunes France
Red Riocha Spain
White Reisling Germany
whovian@phy504:~/host$ rev wine.txt
ynamreG gnilsieR etihW
niapS ahcoiR deR
ecnarF senuaeB deR
```

# Input/Output redirection

Remember command **cat** (copy input to the output).

```
whovian@phy504:~/host$ echo hello.txt  
hello.txt  
whovian@phy504:~/host$ cat hello.txt  
Hello world
```

What if we want the file to become the output (terminal input)?

> and >> symbols are two of many I/O type of redirectors

```
whovian@phy504:~/host$ cat hello.txt  
Hwhovian@phy504:~/host$ cat > hello.txt  
Hello World again with I/O redirection      pressed CTRL+D to  
whovian@phy504:~/host$ cat hello.txt          exit terminal input  
Hello World again with I/O redirection  
whovian@phy504:~/host$ cat >> hello.txt  
Now the new sentence will be appended to the end of the file  
whovian@phy504:~/host$ cat hello.txt  
Hello World again with I/O redirection  
Now the new sentence will be appended to the end of the file
```

# Input/Output redirection

< symbol take standard input from file

```
whovian@phy504:~/host$ read TEST < hello.txt
whovian@phy504:~/host$ echo $TEST
Hello World again with I/O redirection
```

Mixing >> and < input / output redirection

```
whovian@phy504:~/host$ more io.txt
This is IO in and out redirection (example)
whovian@phy504:~/host$ more hello.txt
Hello World again with I/O redirection
Now the new sentence will be appended to the end of the file
whovian@phy504:~/host$ cat >> hello.txt < io.txt
whovian@phy504:~/host$ more hello.txt
Hello World again with I/O redirection
Now the new sentence will be appended to the end of the file
This is IO in and out redirection (example)
```

# Lecture 5: Bash Language Part III

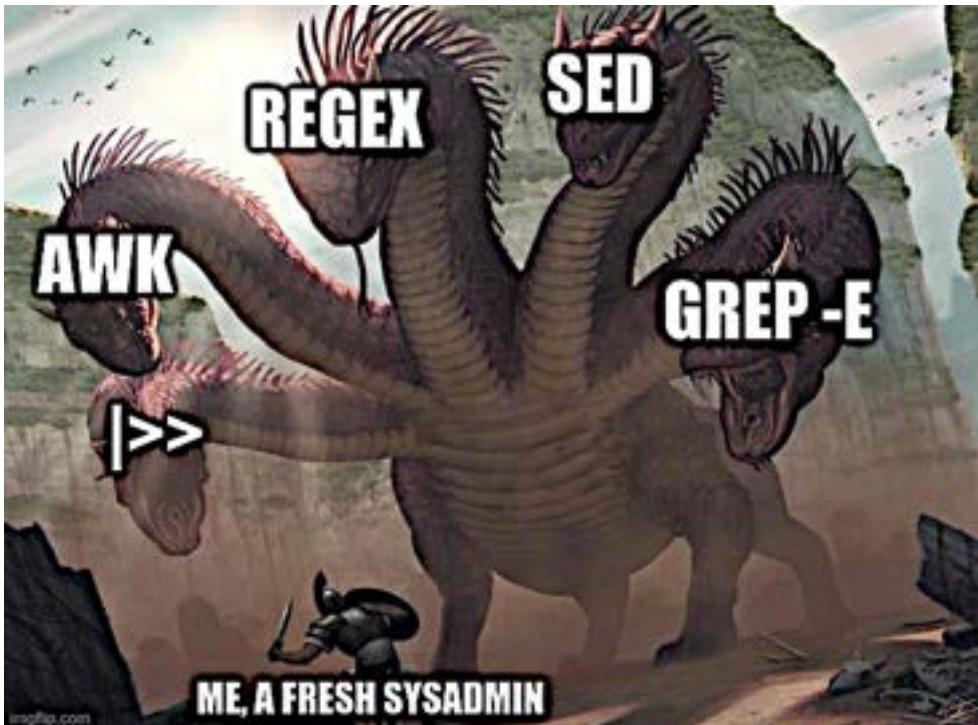
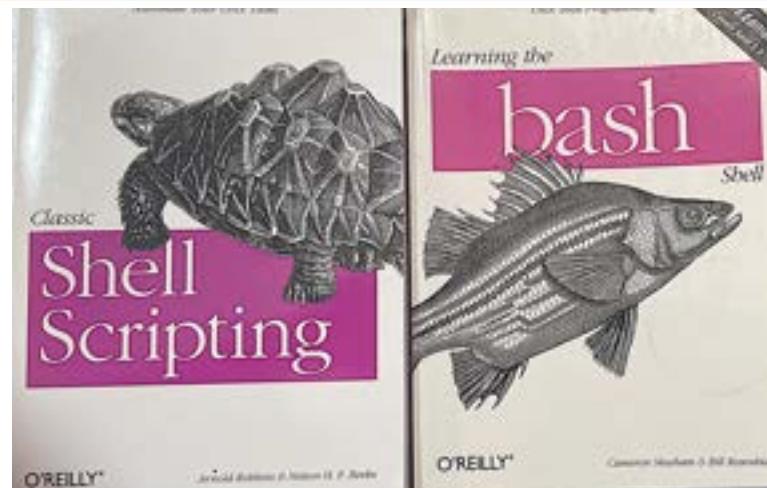


Photo of a new  
Linux user learning  
the command line



## Suggested Literature

Both books are available at  
<https://learning.oreilly.com>



# Basic string manipulation

# Printing: `echo` vs `printf`

`echo`: powerful and for most cast adequate.

However sometimes we need formatted output: `printf`

```
whovian@phy504:~/host$ printf "Hello World\n"
Hello World
whovian@phy504:~/host$ printf "Hello %s\n" world
Hello world
whovian@phy504:~/host$ printf "|%10s|\n" Hello
|      Hello|
whovian@phy504:~/host$ printf "|%-10s|\n" Hello
|Hello      |
whovian@phy504:~/host$ 
whovian@phy504:~/host$ printf "%d + %d = %d\n" 2 2 4
2 + 2 = 4
```

# Printing: echo vs printf

Specifier	Description
%c	ASCII character (prints first character of corresponding argument)
%d	Decimal integer
%i	Same as %d
%e	Floating-point format $([-]d.precisione[+-]dd)$ (see following text for meaning of <i>precision</i> )
%E	Floating-point format $([-]d.precisionE[+-]dd)$
%f	Floating-point format $([-]ddd.precision)$
%g	%e or %f conversion, whichever is shorter, with trailing zeros removed
%G	%E or %f conversion, whichever is shortest, with trailing zeros removed
%o	Unsigned octal value
%s	String
%u	Unsigned decimal value
%x	Unsigned hexadecimal number; uses a-f for 10 to 15
%X	Unsigned hexadecimal number; uses A-F for 10 to 15
%%	Literal %

# Printing: echo vs printf

HW: backlash “\” has a special meaning on printf

Table 7-1. printf escape sequences

Sequence	Description		
		\t	Horizontal tab.
\b	Backspace.	\v	Vertical tab.
\n	Newline.	\	A literal backslash character.

```
whovian@phy504:~/host$ printf "a string, no processing: <%s>\n" "A\nB"
a string, no processing: <A\nB>
whovian@phy504:~/host$ printf "a string, with processing: <%b>\n" "A\n\nB"
a string, with processing: <A
B>
whovian@phy504:~/host$ printf "a string, with processing: <%b>\n" "A\tB"
a string, with processing: <A    B>
```



note the \n at the end of the print statement

# Printing: echo vs printf

Conversion	Precision means
%d, %i, %o, %u, %x, %X	The minimum number of digits to print. When the value has fewer digits, it is padded with leading zeros. The default precision is 1.
%e, %E	The minimum number of digits to print. When the value has fewer digits, it is padded with zeros after the decimal point. The default precision is 6. A precision of 0 inhibits printing of the decimal point.
%f	<small>The Full Story on printf</small> of digits to the right of the decimal point.
%g, %G	The maximum number of significant digits.

```
whovian@phy504:~/host$ printf "%.2f\n" 123.4567
123.46
whovian@phy504:~/host$ printf "%.5G\n" 123.4567
123.46
whovian@phy504:~/host$ printf "%.2G\n" 123.4567
1.2E+02
```

If your integral has an absolute numerical error of 0.01, it makes no sense to print the results with 15 digits

# Printing: echo vs printf

Conversion	Precision means
%d, %i, %o, %u, %x, %X	The minimum number of digits to print. When the value has fewer digits, it is padded with leading zeros. The default precision is 1.
%e, %E	The minimum number of digits to print. When the value has fewer digits, it is padded with zeros after the decimal point. The default precision is 6. A precision of 0 inhibits printing of the decimal point.
%f	<small>The Full Story on printf</small> of digits to the right of the decimal point.
%g, %G	The maximum number of significant digits.

**printf** allows padding (both on numbers and strings)

The **+** flag prints sign (both positive and negative)

```
whovian@phy504:~/host$ printf "%.5d %.5d\n" -15 15  
-00015 00015  
whovian@phy504:~/host$ printf "%+.5d %+.5d\n" -15 15  
-00015 +00015
```

# Printing: echo vs printf

```
whovian@phy504:~/host$ printf "%5d %.5d\n" -15 15  
-00015 00015  
whovian@phy504:~/host$ printf "%+.5d %+.5d\n" -15 15  
-00015 +00015
```

without dots you pad with spaces instead of zeros

```
whovian@phy504:~/host$ printf "%+.5d\n" 15  
+00015  
whovian@phy504:~/host$ printf "%+5d\n" 15  
+15
```

However here the + counts in the 5.

```
whovian@phy504:~/host$ printf "%5d\n" 15  
15  
whovian@phy504:~/host$ printf "%.5d\n" 15  
00015
```

Basic string manipulation  
with a twist: variables

# Defining variables in Bash

We can define variables that can hold values different than their names

```
whovian@phy504:~$ myvar=this_is_a_long_string_that_does_not_mean_much  
whovian@phy504:~$ echo -e '\n' $myvar '\n'  
this_is_a_long_string_that_does_not_mean_much
```

Variables can hold integers. They can also hold floats

```
whovian@phy504:~$ myvar=10  
whovian@phy504:~$  
whovian@phy504:~$ echo -e '\n' $myvar '\n'
```

10

However, there isn't float arithmetic in pure bash

# Reading user input

`read` prompts a user for input

```
GNU nano 4.8                         example.sh
echo -n 'terminal? '
read TERM
echo "TERM is $TERM"
```

```
whovian@phy504:~/host$ sh example.sh
terminal? Oh YES!
TERM is Oh YES!
```

`read` to an array with `-a` option

```
whovian@phy504:~/host$ read -a people
vivian miranda apple
whovian@phy504:~/host$ echo ${people[2]}
apple
whovian@phy504:~/host$ echo ${people[1]}
miranda
```

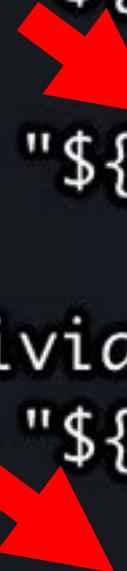
HW:  
research  
about `-n` and  
`-p` options

# Parameter expansion: length of variable's content

How to get the length of the value the variable holds?

`${#variable}`

```
RiverSong:C vivianmiranda$ myvar=1001
RiverSong:C vivianmiranda$ echo -e "${myvar}"
1001
RiverSong:C vivianmiranda$ echo -e "${#myvar}"
4
RiverSong:C vivianmiranda$ myvar=Vivian
RiverSong:C vivianmiranda$ echo -e "${myvar}"
Vivian
RiverSong:C vivianmiranda$ echo -e "${#myvar}"
6
```



# Parameter expansion: default values

Many times, it is important to provide default values / behavior

Operator	Substitution
<code> \${ varname :- word }</code>	If <i>varname</i> exists and isn't null, return its value; otherwise return <i>word</i> .  <b>Purpose:</b> Returning a default value if the variable is undefined.  <b>Example:</b> <code> \${count:-0}</code> evaluates to 0 if <b>count</b> is undefined.

```
whovian@phy504:~/host$ echo ${MYVAR}
```

```
whovian@phy504:~/host$ MYVAR2=${MYVAR:-10}
```

```
whovian@phy504:~/host$ echo ${MYVAR2}
```

```
10
```

```
whovian@phy504:~/host$ echo ${MYVAR}
```

# Parameter expansion: default values

Many times, it is important to provide default values / behavior

Operator	Substitution
<code> \${ varname := word }</code>	If <i>varname</i> exists and isn't null, return its value; otherwise set it to <i>word</i> and then return its value. Positional and special parameters cannot be assigned this way.  <b>Purpose:</b> Setting a variable to a default value if it is undefined.  <b>Example:</b> <code> \${count:=0}</code> sets <b>count</b> to 0 if it is undefined.

```
whovian@phy504:~/host$ echo ${MYVAR}
```

```
whovian@phy504:~/host$ MYVAR2=${MYVAR:=10}
```

```
whovian@phy504:~/host$ echo ${MYVAR2}
```

```
10
```

```
whovian@phy504:~/host$ echo ${MYVAR}
```

```
10
```

# Parameter expansion: error message

Many times, it is important to provide default values / behavior

Operator	Substitution
<code> \${ varname :? message }</code>	If <i>varname</i> exists and isn't null, return its value; otherwise print <i>varname</i> : followed by <i>message</i> , and abort the current command or script (non-interactive shells only). Omitting <i>message</i> produces the default message <b>parameter null or not set</b> .

Syntax of String Operators

**Purpose:** Catching errors that result from variables being undefined.

**Example:** `{count:?"undefined!"}` prints "count: undefined!" and exits if `count` is undefined.

```
whovian@phy504:~/host$ unset MYVAR
```

```
whovian@phy504:~/host$ MYVAR2=${MYVAR:?"error_message"}  
bash: MYVAR: error_message
```

```
whovian@phy504:~$  
whovian@phy504:~$ namedir=${dirname:?"missing directory name"}  
bash: dirname: missing directory name  
whovian@phy504:~$
```

# Convert to Uppercase (bash > 4.0)

define variable with string variable than use ^ or ^^

```
whovian@phy504:~$ name=vivian
whovian@phy504:~$ echo ${name^}
Vivian
whovian@phy504:~$ echo ${name^^}
VIVIAN
whovian@phy504:~$ echo ${name^^v}
ViVian
```

To check your bash version, there is a **default variable**

```
whovian@phy504:~$ echo $BASH_VERSION
5.0.17(1)-release
```

# Convert to LowerCase (bash > 4.0)

define variable with string variable than use , or ,,

```
whovian@phy504:~/host$ name=VIVIAN
whovian@phy504:~/host$ echo ${name,}
vIVIAN
whovian@phy504:~/host$ echo ${name,,}
vivian
whovian@phy504:~/host$ echo ${name,,,V}
vIvIAN
```

To check your bash version, there is a **default variable**

```
whovian@phy504:~$ echo $BASH_VERSION
5.0.17(1)-release
```

# Alias in bash (variable alias)

We can set a variable as alias to our commands

```
RiverSong:Phy504Docker vivianmiranda$ alias phy504='docker run --platfo  
rm linux/amd64 --hostname phy504 -it -p 8080:8888 -v $(pwd):/home/whovi  
an/host/ -v ~/.ssh:/home/whovian/.ssh:ro vivianmiranda/whovian-phy504'  
RiverSong:Phy504Docker vivianmiranda$ phy504
```

```
*****  
*  
* Welcome to Docker Whovian-PHY504 docker image. I hope this VM  
* will help you learn shell, C/C++/Fortran, and Python  
*
```

For example, I alias all machines I access via **ssh**

```
RiverSong:Phy504Docker vivianmiranda$ more ~/.bashrc  
alias midway2='ssh -o ServerAliveInterval=60 viniciusvmb@midway2.rcc.uchicago.edu'  
alias ocelote='ssh -o ServerAliveInterval=60 vivianmiranda@hpc.arizona.edu'  
alias arizona='ssh -o ServerAliveInterval=60 vivianmiranda@amypond.cosmoatuofa.com'  
alias arizona2='ssh -o ServerAliveInterval=60 vivianmiranda@128.196.208.65'
```

# Integer arithmetic

There is no float-point arithmetic. Integer: **\$((...))**

```
whovian@phy504:~$ echo "$(( (365 - $(date +%j) )/7 )) weeks until NYE"
50 weeks until NYE
```

Operator	Meaning
++	Increment by one (prefix and postfix)
-	Decrement by one (prefix and postfix)
+	Plus
-	Minus
*	Multiplication
/	Division (with truncation)
%	Remainder
**	Exponentiation <sup>[10]</sup>

```
whovian@phy504:~$ date
Sun 15 Jan 2023 09:23:48 PM UTC
whovian@phy504:~$ date +%j
015
```

## Increment by one

Because of the use of **\$()**, we could have defined **x=2** without declare

```
whovian@phy504:~$ declare -i x=2
whovian@phy504:~$ echo $x
2
whovian@phy504:~$ echo $((x++))
2
whovian@phy504:~$ echo $((++x))
4
```

# Why **declare -i**?

## **declare** – set variable attributes

Option	Meaning	Option	Meaning
-a	The variables are treated as arrays	-i	The variables are treated as integers
-f	Use function names only	-r	Makes the variables read-only
-F	Display function names without definitions	-x	Marks the variables for export via the environment

```
whovian@phy504:~/host$ declare -i val3=12 val4=5
whovian@phy504:~/host$ declare -i results2
whovian@phy504:~/host$ results2=val3*val4
whovian@phy504:~/host$ echo $results2
60
```

```
whovian@phy504:~/host$ val1=12
whovian@phy504:~/host$ val2=5
whovian@phy504:~/host$ result1=val1*val2
whovian@phy504:~/host$ echo $result1
val1*val2
```

# Positional Parameters

Built-in variables. They have names 1, 2, 3...

Two special variables **@** and **\*** Positional parameter: **0**

The number of positional parameter: **#**

**shift** – move parameters (and delete the first)

```
#!/bin/bash

echo -en "Show all parameters: $@ \n"
echo -en "Show first parameter: $1 \n"
echo -en "Show the number of parameters $# \n"

echo -e "\nThis is going to be interesting now \n"

echo -en "Show first parameter: $1. Number of parameters $# \n"
shift
echo -en "Show second parameter: $1. Number of parameters $# \n"
shift
echo -en "Show third parameter: $1. Number of parameters $# \n"
```

# Positional Parameters

Built-in variables. They have names **1, 2, 3...**

Two special variables **@** and **\*** Positional parameter: **0**

The number of positional parameter: **#**

**shift – move parameters (and delete the first)**

```
whovian@phy504:~/host$ bash positional.sh vivian miranda vania
Show all parameters: vivian miranda vania
Show first parameter: vivian
Show the number of parameters 3
```

This is going to be interesting now

```
Show first parameter: vivian. Number of parameters 3
Show second parameter: miranda. Number of parameters 2
Show third parameter: vania. Number of parameters 1
```

# Lecture 6: Bash Language Part IV

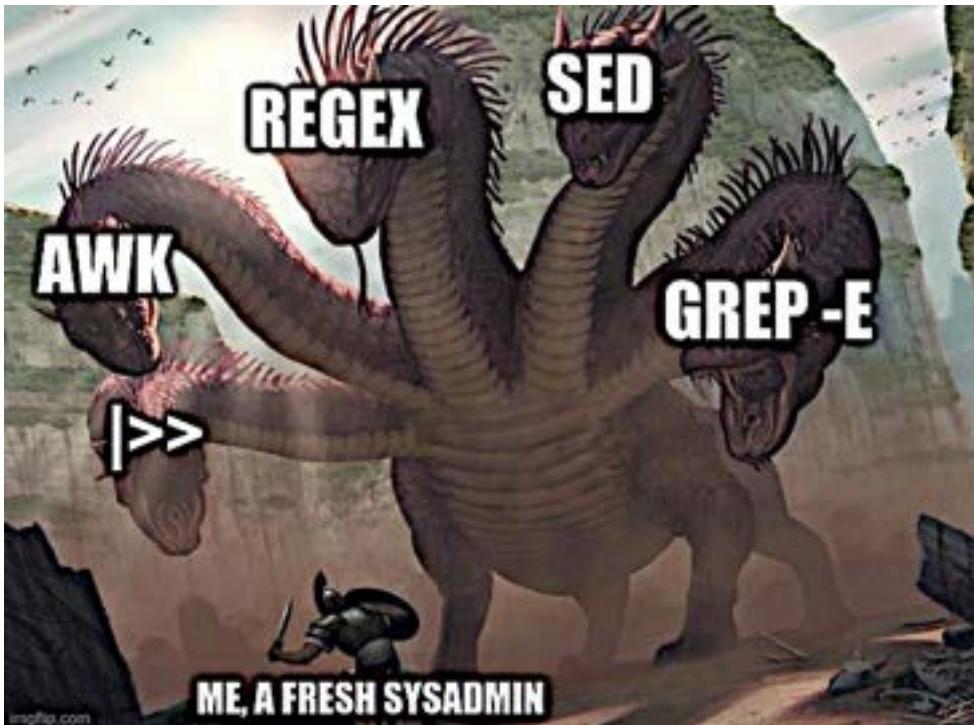
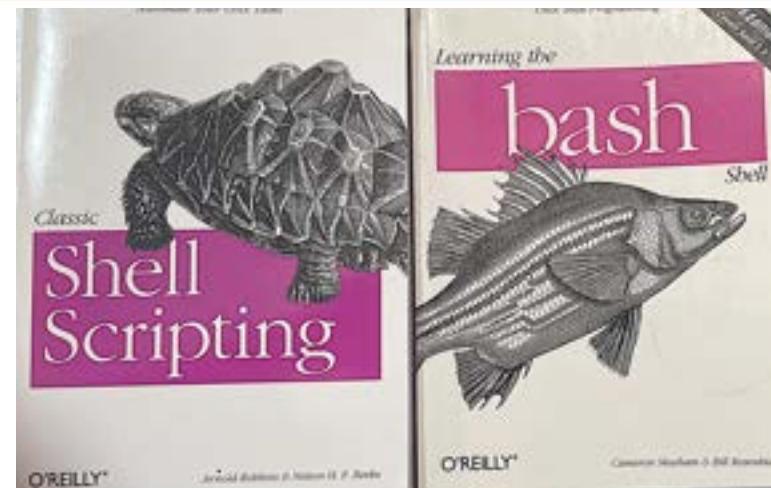


Photo of a new  
Linux user learning  
the command line



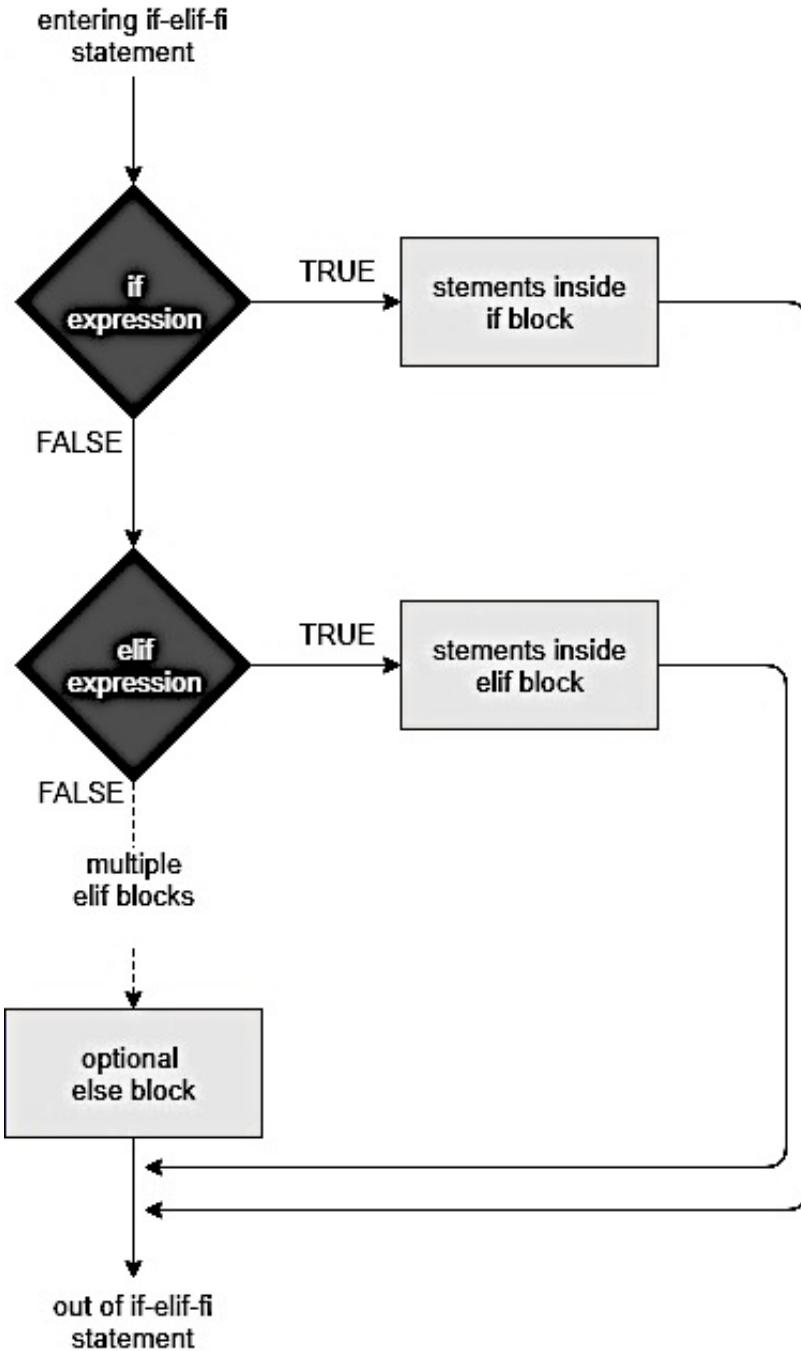
## Suggested Literature

Both books are available at  
<https://learning.oreilly.com>



# if/elif/else

## statements



```
GNU nano 4.8
#!/bin/bash

n=2

if [ $n -eq 1 ]; then
    echo value of n is 1
elif [ $n -eq 2 ]; then
    echo value of n is 2
else
    echo value of n is other than 1 and 2
fi
```

```
whovian@phy504:~/host$ bash ifelse.sh
value of n is 2
```

# Comparing Strings

How can we know if two strings are the same (case sensitive)

GNU nano 4.8

```
#!/bin/bash
s1="HELLO WORLD"
s2="Hello World"
if [ "$s1" = "$s2" ]; then
    echo "s1 and s2 are the same"
else
    echo "Not Same"
fi
```

Question with  
T/F possible  
answers

This runs if  
True

This is the first time  
we also introduce  
if/else statements

Be careful

a = 2

is assignment while

“\$a” = “\$b”

can be string  
comparison

```
whovian@phy504:~/host$ bash sequal.sh
Not Same
```

HW: use the operator != which returns T if strings are not equal

# Comparing Strings

How can we know if one strings is “less than” than other

```
GNU nano 4.8
#!/bin/bash
s1="HELLO WORLD"
s2="HELLO"

if [[ "$s1" < "$s2" ]]; then
    echo "s1 is less than s2"
elif [[ "$s1" > "$s2" ]]; then
    echo "s2 is less than s1"
else
    echo "s1 and s2 are equal"
fi
```

Many interesting commands  
in this script

- **if/elif/else** statements
- Lexicographically comparisons (dictionary)
- double brackets [[ ]] vs []

```
whovian@phy504:~/host$ bash sless.sh
s2 is less than s1
```

GNU nano 4.8

```
#!/bin/bash
```

## Code from my research

```
if [ -z "${ROUTDIR}" ]; then
    echo 'ERROR ROOUTDIR not defined'
    return
fi
if [ -z "${CXX_COMPILER}" ]; then
    echo 'ERROR CXX_COMPILER not defined'
    return
fi
if [ -z "${C_COMPILER}" ]; then
    echo 'ERROR C_COMPILER not defined'
    return
fi
if [ -z "${MAKE_NUM_THREADS}" ]; then
    echo 'ERROR MAKE_NUM_THREADS not defined'
    return
fi

# ----- COMPILE COSMOLIKE -----
cd $ROOUTDIR/projects/des_y3/interface
```

To be, or not  
to be, that is  
the question

The **-z** and **-n**  
comparison operators  
test if a string is null

If your script depends  
on a key being set,  
use if/else with **-n**  
and **-z** operators to  
check that

# Comparing Strings: summary

*Table 5-1. String comparison operators*

Operator	True if...
<code>str1 = str2</code> <sup>[4]</sup>	<code>str1</code> matches <code>str2</code>
<code>str1 != str2</code>	<code>str1</code> does not match <code>str2</code>
<code>str1 &lt; str2</code>	<code>str1</code> is less than <code>str2</code>
<code>str1 &gt; str2</code>	<code>str1</code> is greater than <code>str2</code>
<code>-n str1</code>	<code>str1</code> is not null (has length greater than 0)
<code>-z str1</code>	<code>str1</code> is null (has length 0)

1. 1. Digits
2. 2. Uppercase letters
3. 3. Lowercase letters

< and > are dictionary comparison

# if / elif / else statements

## if / elif / else with file attribute checking (super useful)

Operator	True if...	-f <i>file</i>	<i>file</i> exists and is a regular file (i.e., not a directory or other special type of file)
-a <i>file</i>	<i>file</i> exists	-r <i>file</i>	You have read permission on <i>file</i>
-d <i>file</i>	<i>file</i> exists and is a directory	-s <i>file</i>	<i>file</i> exists and is not empty
-e <i>file</i>	<i>file</i> exists; same as - a	-w <i>file</i>	You have write permission on <i>file</i>

```
whovian@phy504:~/host$ bash ifelse2.sh nofile.sh  
file nofile.sh does not exist.
```

```
#!/bin/bash  
  
if [ ! -e "$1" ]; then  
    echo "file $1 does not exist."  
    exit 1  
fi
```

! opposite positional parameters

Built-in positional parameters. They have names 1, 2, 3...

# if / elif / else statements

Integer comparison have different operators

Table 5-3. Arithmetic test operators

Test	Comparison	-eq	Equal	-gt	Greater than
-lt	Less than	-ge	Greater than or equal	-ne	Not equal
-le	Less than or equal				

```
GNU nano 4.8
#!/bin/bash

if [[ 6 > 57 ]]; then
    echo "Dictionary order"
fi

if [[ 57 -gt 6 ]]; then
    echo "Arithmetic order"
fi
```

```
whovian@phy504:~/host$ bash ifelse.sh
Dictionary order
Arithmetic order
```

# if / elif / else statements

if / elif / else with logical operators (**&&** equals and), (**||** equals or)

The syntax *statement1 && statement2* means, “execute *statement1*, and if its exit status is 0, execute *statement2*.”

The syntax *statement1 || statement2* is the converse: it means, “execute *statement1*, and if its exit status is *not* 0, execute *statement2*.”

```
#!/bin/bash

if [[ $1 -gt 3 && $1 -gt 4 ]]
then
    echo "Yes (&&): $1"
fi

if [[ $1 -gt 3 || $1 -gt 5 ]]
then
    echo "Yes (||): $1"
fi
```

```
whovian@phy504:~/host$ bash ifelse3.sh 4
Yes (&&): 4
whovian@phy504:~/host$ bash ifelse3.sh 5
Yes (&&): 5
Yes (||): 5
```

# Loops: arithmetic **for** loops

The form of an arithmetic for loop is similar to those found in C/C++

```
for (( initialisation ; ending condition ; update ))  
do  
    statements...  
done
```

```
GNU nano 4.8  
#!/bin/bash  
  
for (( i=1; i <= 5; i++ ))  
do  
    for (( j=1; j <=5; j++ ))  
    do  
        echo -ne "$(( j*i ))\t"  
    done  
    echo  
done
```

Arithmetic for loop useful when dealing with arrays

```
whovian@phy504:~/host$ bash forloop.sh  
1      2      3      4      5  
2      4      6      8      10  
3      6      9      12     15  
4      8      12     16     20  
5      10     15     20     25
```

# Loops: arithmetic **for** loops x2

Loop through a range of numbers with a step or a simple manual list

```
GNU nano 4.8
#!/bin/bash
for i in {0..10..2}; do
    echo $i
done
```



```
GNU nano 4.8
#!/bin/bash
for i in 1 2 3 4 5; do
    echo $i
done
```

```
whovian@phy504:~/host$ bash forloop4.sh
0
2
4
6
8
10
```

```
whovian@phy504:~/host$ bash forloop6.sh
1
2
3
4
5
```

The range and list can be a list of strings as well!

# Loops: **for** loops (iteration)

You can loop over files in a directory

```
whovian@phy504:~/host$ echo $PATH  
/opt/conda/bin /usr/local/nvidia/bin /usr/local/cuda/bin  
/usr/local/sbin /usr/local/bin /usr/sbin /usr/bin /sbin  
/bin
```

```
GNU nano 4.8  
#!/bin/bash  
  
IFS=:  
  
for dir in $PATH  
do  
    ls -ld $dir  
done
```

```
for name [in list]  
do  
    statements that can use  
    $name...  
done
```

IFS determines how Bash recognizes word boundaries while splitting a sequence of character strings

The default value of **IFS** is a three-character string comprising a space, tab, and newline:

# Loops: for loops (iteration)

You can loop over files in a directory

```
#!/bin/bash
```

```
for file in ~/host/*; do
  if [ -f "$file" ]; then
    echo "$file is a regular file"
  fi
  if [ -d "$file" ]; then
    echo "$file is a directory"
  fi
done
```

wildcard

(means every file)

-f checks if the  
file is a regular  
file (not a  
symbolic link)

```
whovian@phy504:~/host$ bash forloop3.sh
/home/whovian/host/2 is a regular file
/home/whovian/host/address_list_backup.txt is a regular file
/home/whovian/host/address_list.txt is a regular file
/home/whovian/host/C is a directory
```

# Loops: for loops (iteration)

You can loop over files in a directory recursively using find

```
#!/bin/bash

x=$(find . -type f) ← command substitution

for file in $x; do
    if [ -f "$file" ]; then
        echo "$file is a regular file"
    fi
done

echo -e "\n\nAnother way to write the same loop\n\n"

for file in $(find . -type f); do
    if [ -f "$file" ]; then
        echo "$file is a regular file"
    fi
done
```

# while loops

while loops run until certain condition is True

```
while [ condition ]
do
    commands
done
```

```
GNU nano 4.8
#!/bin/bash

# Initialize the counter
n=1
while [ $n -le 5 ]
do
    echo "Running $n time"
    # Increment the value of n by 1
    (( n++ ))
done
```

```
whovian@phy504:~/host$ bash whileloop.sh
Running 1 time
Running 2 time
Running 3 time
Running 4 time
Running 5 time
```

# while loops x2

Warning: **while** [True]

never stops

(unless we **break** it!)

How **break** works?

```
#!/bin/bash

# Initialize the counter
n=1
# Iterate the loop for 10 times
while [ $n -le 10 ]
do
    if [[ $n == 6 ]]
    then
        echo "terminated"
        break
    fi
    echo "Position: $n"
    (( n++ ))
done
```

```
#           echo "terminated"
#           break
#           (( n++ ))
#           continue
```

```
whovian@phy504:~/host$ bash whileloop2.sh
Position: 1
Position: 2
Position: 3
Position: 4
Position: 5
terminated
```

**break** vs **continue**

```
Position: 5
terminated
Position: 7
```



# while loops x3 – nested while loops

```
#!/bin/bash

n=1
while [ $n -le 3 ]
do
    echo "Position: n=$n"
    m=1
    while [ $m -le 3 ]
    do
        echo -ne "Position: m=$m \t"
        if [[ $m == 2 ]]
        then
            echo -ne "terminated \n"
            break
        fi
        (( m++ ))
    done
    (( n++ ))
done
```

How **break** works?

**break** in the inner loop  
exit only the inner loop

```
whovian@phy504:~/host$ bash whileloop3.sh
Position: n=1      Position: m=1      Position: m=2      terminated
Position: n=2      Position: m=1      Position: m=2      terminated
Position: n=3      Position: m=1      Position: m=2      terminated
```

# case loops

```
#!/bin/bash

echo -n "Enter the name of a country: "
read COUNTRY

echo -n "The official language of $COUNTRY is "

case $COUNTRY in
    Lithuania)
        echo "Lithuanian"
        ;;
    Romania | Moldova)
        echo "Romanian"
        ;;
    Italy | "San Marino" | "Vatican City")
        echo "Italian"
        ;;
    *)
        echo "unknown"
        ;;
esac
```

You can use logical operators **&&** and **||**



```
whovian@phy504:~/host$ bash caseloop.sh
Enter the name of a country: Brazil
The official language of Brazil is unknown
```

# while + case loops (option processing)

```
#!/bin/bash
# set flag vars to empty
file= verbose= quiet=
while [ $# -gt 0 ]           Loop until no args left
do
    case $1 in
        -f)   file=$2
              shift
              ;;
        -v)   verbose=true
              quiet=
              ;;
        --)   shift          #By convention, -- ends options
              break
              ;;
        -*)  echo $0: $1: unrecognized option >&2
              ;;
        *)   break          #Nonoption argument, break while loop
              ;;
    esac
    shift                      #Set up for next iteration
done
```

Number of positional parameters



There is one limitation. No implementation of compound options like –fg. The preferred method for options is with `getopts`

# while + case + getopt (option processing)

getopt allows multiple options in the same dash, e.g., -fv

```
#!/bin/bash
file=
while getopt "fvg:" optname
do
    case ${optname} in
        f) echo "-f option!"
            ;;
        v) echo "-v option!"
            ;;
        g) file=${OPTARG}
            echo "-g option! File=${file}"
            ;;
        :) echo -e "Error: -${OPTARG} requires an argument\r"
            exit 1
            ;;
        *) echo "Option does not exist"
            break
            ;;
    esac
done
```

Start with :, then for each option that needs an argument, write an :

Variable OPTARG provides the arguments

:) = whenever an argument is not provided

You don't need the dash

no need to use shift

# while + case + getopts (option processing)

```
whovian@phy504:~/host$ bash whileloop5.sh -f  
-f option!  
whovian@phy504:~/host$ bash whileloop5.sh -v  
-v option!  
whovian@phy504:~/host$ bash whileloop5.sh -fv  
-f option!  
-v option!  
whovian@phy504:~/host$ bash whileloop5.sh -g vivian.txt  
-g option! File=vivian.txt  
whovian@phy504:~/host$ bash whileloop5.sh -g  
Error: -g requires an argument
```

mixing **getopts** and positional parameters is quite advanced  
(I won't ask that – see StackOverflow [answer](#))

**getopts** is to deal with options using dash

Positional parameters do not need to come with a dash

# Lecture. 7: Shell Part V

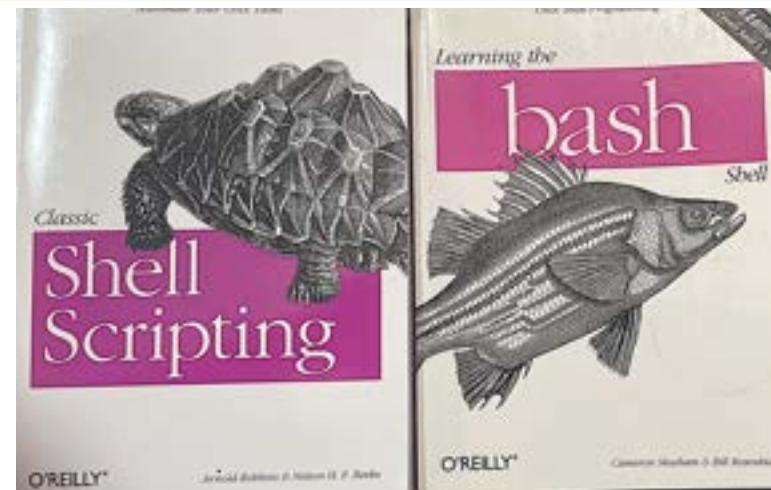


Photo of a new  
Linux user learning  
the command line



## Suggested Literature

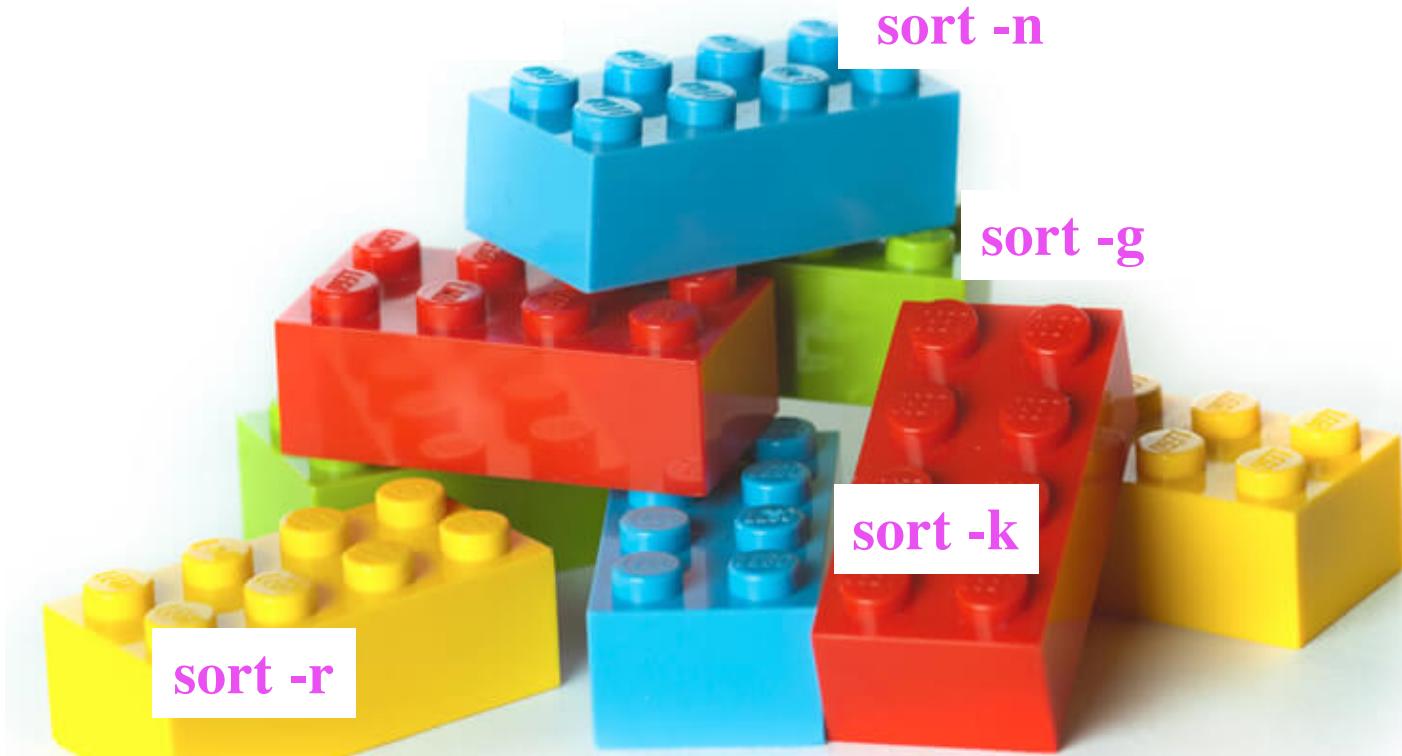
Both books are available at  
<https://learning.oreilly.com>



# Command line options – coloring Lego

Command line options provide additional functionality to bash programs (“coloring Lego Bricks”)

---



```
whovian@phy504:~/host$ more singers.txt
4 Avril_Lavigne.
10 Rihanna.
2 Lady_Gaga.
5 Usher
14 Shakira
1 Justin_Timberlake.
20 Billie_Joe_Armstrong
3 Taylor_Swift.
whovian@phy504:~/host$ sort -n singers.txt
1 Justin_Timberlake.
2 Lady_Gaga.
3 Taylor_Swift.
4 Avril_Lavigne.
5 Usher
10 Rihanna.
14 Shakira
20 Billie_Joe_Armstrong
whovian@phy504:~/host$ sort -nr singers.txt
20 Billie_Joe_Armstrong
14 Shakira
10 Rihanna.
5 Usher
4 Avril_Lavigne.
3 Taylor_Swift.
2 Lady_Gaga.
1 Justin_Timberlake.
```

# Sorting integers with **sort**

File with singers and the  
number of albums you own  
from these singers

**-n** option: interpret column  
as numbers (not string)

**-r** reverse order

What if the numbers we  
want to sort are not in the  
first order

```
whovian@phy504:~/host$ more singers2.txt
4 Avril_Lavigne 100
10 Rihanna 50
2 Lady_Gaga 300
5 Usher 10
14 Shakira 1000
1 Justin_Timberlake 25
20 Billie_Joe_Armstrong 250
3 Taylor_Swift 500
whovian@phy504:~/host$ sort --key 3 -n singers2.txt
5 Usher 10
1 Justin_Timberlake 25
10 Rihanna 50
4 Avril_Lavigne 100
20 Billie_Joe_Armstrong 250
2 Lady_Gaga 300
3 Taylor_Swift 500
14 Shakira 1000
whovian@phy504:~/host$ sort --key 3 -nr singers2.txt
14 Shakira 1000
3 Taylor_Swift 500
2 Lady_Gaga 300
20 Billie_Joe_Armstrong 250
4 Avril_Lavigne 100
10 Rihanna 50
1 Justin_Timberlake 25
5 Usher 10
```

# Sorting integers with **sort**

Added a third  
column with the  
amount of money  
you spent on  
singer's albums

# Sorting dictionary ordering

```
whovian@phy504:~/host$ sort --key 1 sorting2.txt
a   1   1.0
a   13  2.1
a   4   0.001
a   6   0.01
a   8   0.35
b   12  3.1
b   2   0.1
b   7   0.01
b   9   2.3
c   10  0.1
c   11  1.0
c   3   0.3
c   5   0.5
whovian@phy504:~/host$ sort --key 1,1d --key 3,3g sorting2.txt
a   4   0.001
a   6   0.01
a   8   0.35
a   1   1.0
a   13  2.1
b   7   0.01
b   2   0.1
b   9   2.3
b   12  3.1
c   10  0.1
c   3   0.3
c   5   0.5
c   11  1.0
```

- The argument `-k1,1d` means “create a key starting at column 1 and ending at column 1 in dictionary order”
- For the last key you could have written `--key 3`

# Sorting with different separators

```
whovian@phy504:~/host$ more sorting3.txt
a,1,1.0
b,2,0.1
c,3,0.3
a,4,0.001
c,5,0.5
a,6,0.01
b,7,0.01
a,8,0.35
b,9,2.3
c,10,0.1
c,11,1.0
b,12,3.1
a,13,2.1
```



```
whovian@phy504:~/host$ sort -t"," --key 1,1d --key 3,3g sorting3.txt
a,4,0.001
a,6,0.01
a,8,0.35
a,1,1.0
a,13,2.1
b,7,0.01
b,2,0.1
b,9,2.3
b,12,3.1
c,10,0.1
c,3,0.3
c,5,0.5
c,11,1.0
```

# Sorting with floating numbers

Did you notice the **-g** option on last slide?

- When dealing with float point **-g** is preferable
- When using scientific notation **-g** is mandatory
- [Awesome stackoverflow answer with in-depth explanation](#)

```
whovian@phy504:~/host$ printf '%s\n' 0.1 10 1e-2 | sort -n
0.1
1e-2
10
whovian@phy504:~/host$ printf '%s\n' 0.1 10 1e-2 | sort -g
1e-2
0.1
10
```

Why not using **-g** all the time instead of **-n**? Speed!

# Stable Sorting

Stable sort algorithms sort equal elements in the same order that they appear in the input.

```
whovian@phy504:~$ more stablesort.txt
1 4
1 3
1 2
1 1
2 1
2 2
2 3
whovian@phy504:~$ sort stablesort.txt -s -k1,1
1 4
1 3
1 2
1 1
2 1
2 2
2 3
whovian@phy504:~$ sort stablesort.txt -k1,1
1 1
1 2
1 3
1 4
2 1
2 2
2 3
```

# Piping: bash scripts is like playing Lego



We have many small pieces (programs).

From “gluing” of programs we build pipelines



# Piping commands

Example: piping **sort** and **head** commands with |

```
whovian@phy504:~/host$ sort -n singers.txt
```

```
1 Justin_Timberlake.
```

```
2 Lady_Gaga.
```

```
3 Taylor_Swift.
```

```
4 Avril_Lavigne.
```

```
5 Usher
```

```
10 Rihanna.
```

```
14 Shakira
```

```
20 Billie_Joe_Armstrong
```

```
whovian@phy504:~/host$ head -n 3 singers.txt
```

```
4 Avril_Lavigne.
```

```
10 Rihanna.
```

```
2 Lady_Gaga.
```

```
whovian@phy504:~/host$ sort -n singers.txt | head -n 3
```

```
1 Justin_Timberlake.
```

```
2 Lady_Gaga.
```

```
3 Taylor_Swift.
```

Lego analogy starts to  
make sense now  
building blocks



# Removing duplicate words

Piping is powerful: lets pip **sort** and **uniq** command

```
whovian@phy504:~/host$ more portugues-numbers
quatro
um
seis
sete
dois
dez
cinco
tres
nove
dez
quatro
oito
oito
oito
```

Portuguese  
numbers between  
one and ten with  
repetition

**uniq** command filters  
adjacent lines from input

- **-d** print duplicates
- **-u** only print unique lines
- **-c** count the records

Because of the word adjacent,  
we to first sort the lines

```
whovian@phy504:~/host$ sort portugues-numbers | uniq -u
cinco
dois
nove
seis
sete
tres
um
```

# Removing duplicate words

Piping is extremely powerful: lets pip **sort** and **uniq** command  
**uniq** command **filters *adjacent*** lines from input

- **-d** removes duplicates
- **-u** only print unique lines
- **-c** count the records

```
whovian@phy504:~/host$ sort portugues-numbers | uniq -d
dez
oito
quatro
```

```
whovian@phy504:~/host$ sort portugues-numbers | uniq -c
1 cinco
2 dez
1 dois
1 nove
3 oito
2 quatro
1 seis
1 sete
1 tres
1 um
```

# Counting lines, words and chars

**wc** command counts bytes / chars / words or lines

- **-c** count bytes
- **-w** count words
- **-l** count lines

```
whovian@phy504:~/host$ echo This is a test of the emergency broadcast system | wc -l
1
whovian@phy504:~/host$ echo This is a test of the emergency broadcast system | wc -w
9
whovian@phy504:~/host$ echo This is a test of the emergency broadcast system | wc -c
48
whovian@phy504:~/host$ echo This is a test of the emergency broadcast system | wc
    1      9      48
```

**wc** accepts one or multiple files as input – producing one-line reports for each, followed by summary report

```
whovian@phy504:~/host$ wc /etc/passwd /etc/group
 25   35 1324 /etc/passwd
 45   45  569 /etc/group
 70   80 1893 total
```

# Reformatting file output with **pr**

**pr** command paginate or columnate file(s) for printing

Few Options:   **-d** double space   **-wNUM** width   **-l** count lines

```
whovian@phy504:~/host$ more table.txt | pr --columns=3 -t -w80
2      3      4      5  12     18     24      30  22     33     44      55
4      6      8      10 14     21     28      35  24     36     48      60
6      9      12     15 16     24     32      40  26     39     52      65
8     12     16     20 18     27     36      45  28     42     56      70
10    15     20     25 20     30     40      50  30     45     60      75
whovian@phy504:~/host$ more table.txt | pr --columns=3 -t -w100
2      3      4      5      12     18     24      30      22     33     44      55
4      6      8      10     14     21     28      35      24     36     48      60
6      9      12     15     16     24     32      40      26     39     52      65
8     12     16     20     20     27     36      45      28     42     56      70
10    15     20     25     20     30     40      50      30     45     60      75
whovian@phy504:~/host$ more table.txt | pr --columns=3 -t -w100 -d
2      3      4      5      12     18     24      30      22     33     44      55
4      6      8      10     14     21     28      35      24     36     48      60
6      9      12     15     16     24     32      40      26     39     52      65
8     12     16     20     20     27     36      45      28     42     56      70
10    15     20     25     20     30     40      50      30     45     60      75
```

# Removing CAPS in a text

**tr SET1 [SET2]** is a powerful command that translate, squeeze, and/or delete chars from standard input, writing to standard output.

- **-c** use the complement of **SET1**
- **-d** delete characters in **SET1**, do not translate
- **-s** replace sequence of a repeated chars with a single occurrence
- **-t** first truncate **SET1** to length of **SET1**

```
whovian@phy504:~/host$ echo Vivian viVIAN viViAN VIVIAN | tr A-Z a-z
vivian vivian vivian vivian
```

## Replacing nonletters (ex: spaces) with newline

```
whovian@phy504:~/host$ echo Vivian viVIAN viViAN VIVIAN | tr -cs A-Za-z' '\n'
Vivian
viVIAN
viViAN
VIVIAN
```

# Application counting words on Hamlet

An illustrative example of the power of shell commands

```
GNU nano 4.8
#!/bin/bash

tr -cs A-Za-z\\' '\\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -k1,1nr -k2 |
sed ${1:-25}q
```

- Replace nonletters with newlines
- Map uppercase to lowercase
- Sort the words in ascending order
- Eliminate duplicates, showing their counts
- Sort by descending count, then by ascending word
- only line we don't understand (next class)

# Application counting words on Hamlet

## How many unique words on Hamlet?

```
whovian@phy504:~/host$ bash wf.sh 99999999999999 < hamlet.txt | wc -l  
4562
```

## Least-frequent words on Hamlet?

```
whovian@phy504:~/host$ bash wf.sh 9999999 < hamlet.txt | tail -n 12 | pr --columns=4 -t -w80  
1 yaw          1 yeoman        1 yielding      1 younger  
1 yawn         1 yesterday     1 yon           1 yourselves  
1 yeastly      1 yesternight   1 yond          1 zone
```

## Most-frequent words on Hamlet?

```
whovian@phy504:~/host$ bash wf.sh 12 < hamlet.txt | pr --columns=4 -t -w80  
1118 the          677 of          559 you         466 in  
987 and          635 i           516 my          420 it  
747 to           566 a           474 hamlet     407 that
```

# Application counting words on Hamlet

How many unique words? (uses regex)

```
whovian@phy504:~/host$ bash wf.sh 999999 < hamlet.txt | grep -c '^ *1.'  
2847  
whovian@phy504:~/host$ █
```

Regex (regular expressions) is the topic of next class

# Lecture 8: Shell Part VI, last Lecture



Photo of a new  
Linux user learning  
the command line



## Suggested Literature

Both books are available at  
<https://learning.oreilly.com>



# **sed** – string substitution

**sed** is a small power string editor program.

In combination with **regular expressions**,  
**sed** has saved my research a couple of times

Let's start simple. Consider this file

```
GNU nano 4.8
John Daggett, 341 King Road, Plymouth MA
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury MA
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston MA
█
```

Nice documentation: <https://www.gnu.org/software/sed/manual/>

# **sed** – string substitution

**sed** for **string substitution:**



```
whovian@phy504:~/host$ sed 's/MA/Massachusetts/' address_list.txt
John Daggett, 341 King Road, Plymouth Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston Massachusetts
```



- note the / (3 of them all required) ‘**s/oldtext/newtext/**’
- Note that sed wrote the altered test in the terminal. To do in-place modifications we need to use the **-i** option

# sed – string substitution

what about multiple substitutions? Use `-e`

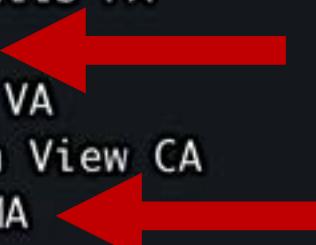
```
wovian@phy504:~/host$ sed -e 's/ MA/, Massachusetts/'  
-e 's/ PA/, Pennsylvania/' address_list.txt  
John Daggett, 341 King Road, Plymouth, Massachusetts  
Alice Ford, 22 East Broadway, Richmond VA  
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK  
Terry Kalkas, 402 Lans Road, Beaver Falls, Pennsylvania  
Eric Adams, 20 Post Road, Sudbury, Massachusetts  
Hubert Sims, 328A Brook Road, Roanoke VA  
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA  
Sal Carpenter, 73 6th Street, Boston, Massachusetts
```

- Look the space before MA. Be careful with characters between //
- Output redirection to save to a new file `sed -f sedscr list > newlist`

# sed – string substitution

You can limit the lines where sed can replace strings

```
whovian@phy504:~/host$ sed -e '1,4s/ MA/, Massachusetts/' address_list.txt
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury MA
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston MA
```



```
whovian@phy504:~/host$ sed -e '1,5s/ MA/, Massachusetts/' address_list.txt
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury, Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston MA
```



# **sed** – string substitution

**sed** can replace only lines that contains some key string

```
whovian@phy504:~/host$ sed -e '/Road/s/ MA/, Massachusetts/' address_list.txt
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury, Massachusetts
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston MA
```

sed only replaces **MA** with **, Massachusetts** on lines that contain **Road**

```
whovian@phy504:~/host$ sed -e '/Road/!s/ MA/, Massachusetts/' address_list.txt
John Daggett, 341 King Road, Plymouth MA
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury MA
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston, Massachusetts
```

! = opposite behavior. Replace only on lines that don't contain Road

# **sed** – string substitution

For longer scripts – create a file and use **-f**

```
GNU nano 4.8                                sedscript
s/ MA/, Massachusetts/
s/ PA/, Pennsylvania/
s/ CA/, California/
s/ VA/, Virginia/
s/ OK/, Oklahoma/█ █
```

```
whovian@phy504:~/host$ sed -f sedscript address_list.txt
John Daggett, 341 King Road, Plymouth, Massachusetts
Alice Ford, 22 East Broadway, Richmond, Virginia
Orville Thomas, 11345 Oak Bridge Road, Tulsa, Oklahoma
Terry Kalkas, 402 Lans Road, Beaver Falls, Pennsylvania
Eric Adams, 20 Post Road, Sudbury, Massachusetts
Hubert Sims, 328A Brook Road, Roanoke, Virginia
Amy Wilde, 334 Bayshore Pkwy, Mountain View, California
Sal Carpenter, 73 6th Street, Boston, Massachusetts
```

# **sed** - Suppressing display of input lines

Default operation of **sed** is to output every input line (except with **-n**). With **-n**, each instruction intended to produce an output must contain a print command, **p**.

```
whovian@phy504:~/host$ sed -n -e 's/MA/Massachusetts/p'  
address_list.txt  
John Daggett, 341 King Road, Plymouth Massachusetts  
Eric Adams, 20 Post Road, Sudbury Massachusetts  
Sal Carpenter, 73 6th Street, Boston Massachusetts
```



Option	Description
<b>-e</b>	Editing instruction follows.
<b>-f</b>	Filename of script follows.

without **p**, no output at all  
with **p**, output only modified lines

**-e**      Editing instruction follows.

**-n**

Suppress automatic output of input lines.

# **sed** - Suppressing display of input lines

With **-n**, each instruction intended to produce output must contain a print command, **p**. We can use that to just print lines with (or without given the **!** command) a special word

```
whovian@phy504:~/host$ sed -n '/MA/p' address_list.txt  
John Daggett, 341 King Road, Plymouth MA  
Eric Adams, 20 Post Road, Sudbury MA  
Sal Carpenter, 73 6th Street, Boston MA
```

```
whovian@phy504:~/host$ sed -n '/MA/!p' address_list.txt  
Alice Ford, 22 East Broadway, Richmond VA  
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK  
Terry Kalkas, 402 Lans Road, Beaver Falls PA  
Hubert Sims, 328A Brook Road, Roanoke VA  
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
```



# **sed** with ‘i’ - writing lines

**sed** can be used to insert new text in a file

```
whovian@phy504:~/host$ sed -e '2i Vivian Miranda, Jamaica, New York City, NY' address_list.txt  
John Daggett, 341 King Road, Plymouth MA  
Vivian Miranda, Jamaica, New York City, NY ←  
Alice Ford, 22 East Broadway, Richmond VA  
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK  
Terry Kalkas, 402 Lans Road, Beaver Falls PA  
Eric Adams, 20 Post Road, Sudbury MA  
Hubert Sims, 328A Brook Road, Roanoke VA  
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA  
Sal Carpenter, 73 6th Street, Boston MA
```

**sed** can also be used to just replace a line

```
whovian@phy504:~/host$ sed -e '3c Vivian Miranda, Jamaica, New York City, NY' address_list.txt  
John Daggett, 341 King Road, Plymouth MA  
Alice Ford, 22 East Broadway, Richmond VA  
Vivian Miranda, Jamaica, New York City, NY ←  
Terry Kalkas, 402 Lans Road, Beaver Falls PA  
Eric Adams, 20 Post Road, Sudbury MA  
Hubert Sims, 328A Brook Road, Roanoke VA  
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA  
Sal Carpenter, 73 6th Street, Boston MA
```

**sed** is so versatile. More functionality  
**sed** can be used to delete lines that contain a key word

```
whovian@phy504:~/host$ sed '/MA/d' address_list.txt
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
```

Print the line numbers of lines containing a key word

```
whovian@phy504:~/host$ sed -n '/MA/=/' address_list.txt
1
5
8
```

# **sed** is so versatile. More functionality sed commands summary

The following commands are supported in GNU sed. Some are standard POSIX commands, while other are GNU extensions. Details and examples for each command are in the following sections. (Mnemonics) are shown in parentheses.

**a\**

**text**

Append *text* after a line.

**a** **text**

Append *text* after a line (alternative syntax).

**b** **label**

Branch unconditionally to *label*. The *label* may be omitted, in which case the next cycle is started.

**c\**

**text**

Replace (change) lines with *text*.

**c** **text**

Replace (change) lines with *text* (alternative syntax).

**d**

Delete the pattern space; immediately start next cycle.

and much more...

<https://www.gnu.org/software/sed/manual/sed.html#sed-commands-list>

# **sed** global option caveat substitution commands – the global option **g**

```
whovian@phy504:~/host$ sed -e '2i Vivian Miranda, Jamaica, New York City NY NY'  
address_list.txt | sed -e 's/ NY/, New York/'  
John Daggett, 341 King Road, Plymouth MA  
Vivian Miranda, Jamaica, New York City, New York NY ←  
Alice Ford, 22 East Broadway, Richmond VA  
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK  
Terry Kalkas, 402 Lans Road, Beaver Falls PA  
Eric Adams, 20 Post Road, Sudbury MA  
Hubert Sims, 328A Brook Road, Roanoke VA  
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA  
Sal Carpenter, 73 6th Street, Boston MA
```

without g, sed only  
do one substitution  
per line

```
whovian@phy504:~/host$ sed -e '2i Vivian Miranda, Jamaica, New York City NY NY'  
address_list.txt | sed -e 's/ NY/, New York/g' ←  
John Daggett, 341 King Road, Plymouth MA  
Vivian Miranda, Jamaica, New York City, New York, New York ←  
Alice Ford, 22 East Broadway, Richmond VA  
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK  
Terry Kalkas, 402 Lans Road, Beaver Falls PA  
Eric Adams, 20 Post Road, Sudbury MA  
Hubert Sims, 328A Brook Road, Roanoke VA  
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA  
Sal Carpenter, 73 6th Street, Boston MA
```

# **sed** global option

Replacing tabs with space

First example replaced tabs with letter A (just to check)

```
whovian@phy504:~/host/Phy504Docker$ echo -e "1\t2\t3" | sed '$s/\t/A/g'  
1A2A3  
whovian@phy504:~/host/Phy504Docker$ echo -e "1\t2\t3" | sed '$s/\t/      /g'  
1      2      3
```

# Regular Expressions

Power & useful feature to deal with strings

Regex is so useful when combined with **egrep** and **sed**

```
whovian@phy504:~/host$ egrep --help
Usage: grep [OPTION]... PATTERN [FILE]...
Search for PATTERN in each FILE.
Example: grep -i 'hello world' menu.h main.c
PATTERN can contain multiple patterns separated by newlines.
```

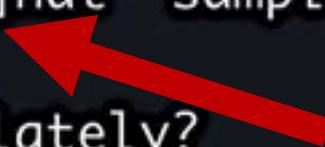
Hey Phy504  
What are you doing?  
Really, what have you been doing lately?  
Heyyyyyy, what is the nature of Physics?  
What is the microphysics of dark energy?  
Xhat is the UV physics of the Universe?  
1hat is the IR physics of the Universe?  
123hat is the EFT physics of the Universe?

Sample text

# Regular Expressions

Regex is so useful when combined with **egrep** and **sed**  
[...] is a metacharacter in Regex

```
whovian@phy504:~/host$ egrep '[Ww]hat' sample_text_egrep.txt
What are you doing?
Really, what have you been doing lately?
Heyyyyyy, what is the nature of Physics?
What is the microphysics_ of dark energy?
```



.

Matches any *single* character except newline

\*

Matches any number (including zero) of the single character (including a character specified by a regular expression) that immediately precedes it.

[...]

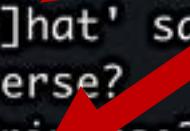
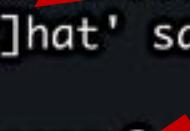
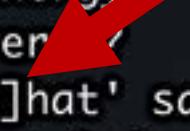
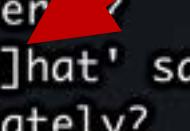
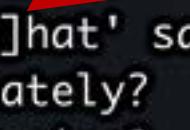
Matches any *one* of the class of characters enclosed between the brackets. A circumflex (^) as first character in-

# Regular Expressions

- \$ As last character of regular expression, matches the end of the line. Matches the end of a string in awk, even if the string contains embedded newlines.
- + Matches one or more occurrences of the preceding regular expression.
- ? Matches zero or one occurrences of the preceding regular expression.
- | Specifies that either the preceding or following regular expression can be matched (alternation).
- ( ) Groups regular expressions.
- \{*n,m\}* Matches a range of occurrences of the single character (including a character specified by a regular expression) that immediately precedes it. \{*n\}\} will match exactly *n* occurrences, \{*n,\}\} will match at least *n* occurrences, and \{*n,m\}\} will match any number of occurrences between *n* and *m*.  
(sed and grep only, may not be in some very old versions.)***
- \ Escapes the special character that follows.

# Regular Expressions

## Metacharacters in Regex

```
whovian@phy504:~/host$ egrep '.hat' sample_text_egrep.txt
What are you doing? 
Really, what have you been doing lately?
Heyyyyyy, what is the nature of Physics?
What is the microphysics of dark energy?
Xhat is the UV physics of the Universe?
1hat is the IR physics of the Universe?
123hat is the EFT physics of the Universe?
whovian@phy504:~/host$ egrep '^[1-9]hat' sample_text_egrep.txt
1hat is the IR physics of the Universe? 
whovian@phy504:~/host$ egrep '[1-9]hat' sample_text_egrep.txt
1hat is the IR physics of the Universe? 
123hat is the EFT physics of the Universe? 
whovian@phy504:~/host$ egrep '[A-Z]hat' sample_text_egrep.txt
What are you doing? 
What is the microphysics of dark energy? 
Xhat is the UV physics of the Universe? 
whovian@phy504:~/host$ egrep '[a-z]hat' sample_text_egrep.txt
Really, what have you been doing lately? 
Heyyyyyy, what is the nature of Physics?
```

# Regular Expressions

Question: Why both expressions matched 123hat?

```
whovian@phy504:~/host$ egrep "[1-9]+hat" sample_text_egrep.txt
1hat is the IR physics of the Universe?
123hat is the EFT physics of the Universe?
whovian@phy504:~/host$ egrep "[1-9]hat" sample_text_egrep.txt
1hat is the IR physics of the Universe?
123hat is the EFT physics of the Universe?
```

The regex **[1-9]hat** matched **3hat**. Let's use the **^** to prove that

```
whovian@phy504:~/host$ egrep "^[1-9]hat" sample_text_egrep.txt
1hat is the IR physics of the Universe?
whovian@phy504:~/host$ egrep "^[1-9]+hat" sample_text_egrep.txt
1hat is the IR physics of the Universe?
123hat is the EFT physics of the Universe?
```

**^** = match from the beginning of the string / line.

So, you need multiple digits (**+**) to go to hat!

# Regular Expressions

Be careful. The position of symbols matter!

```
whovian@phy504:~/host$ egrep '[^1-9]hat' sample_text_egrep.txt
What are you doing?
Really, what have you been doin lately?
Heyyyyyy, what is the nature of Physics?
What is the microphysics of dar energy?
Xhat is the UV physics of the Universe?
```

^K here means all except [1-9]

Metacharacters: second example

GNU nano 4.8	
I can do it	I can't do it
I cannot do it	whovian@phy504:~/host\$ egrep "can[ no']t" sample_text_egrep2.txt
I can not do it	I cannot do it
I can't do it	I can not do it
I cant do it	I can't do it

# Regular Expressions

It is sometimes difficult to match a complete word. If we want to match the pattern “book”, our search will hit the word “book” and “books” but also “bookish,” “handbook,” and “booky.”

```
book 1  
books 2  
bookish, 3  
handbook, 4  
booky. 5  
book? 6  
book, 7  
book! 8
```

```
whovian@phy504:~/host$ egrep 'book' sample_text_egrep3.txt  
book 1  
books 2  
bookish, 3  
handbook, 4  
booky. 5  
book? 6  
book, 7  
book! 8
```

Both wrong

# Regular Expressions

It is sometimes difficult to match a complete word. If we want to match the pattern “book”, our search will hit the word “book” and “books” but also “bookish,” “handbook,” and “booky.”

```
book 1
books 2
bookish, 3
handbook, 4
booky. 5
book? 6
book, 7
book! 8
```

```
whovian@phy504:~/host$ egrep 'books*' sample_text_egrep3.txt
book 1
books 2
whovian@phy504:~/host$ egrep 'book.*' sample_text_egrep3.txt
book 1
books 2
bookish, 3
handbook, 4
booky. 5
book? 6
book, 7
book! 8
whovian@phy504:~/host$ egrep 'book.? ' sample_text_egrep3.txt
book 1
books 2
handbook, 4
book? 6
book, 7
book! 8
```



0 or 1 occurrences of the preceding character

# Regular Expressions

It is sometimes difficult to match a complete word. If we want to match the pattern “book”, our search will hit the word “book” and “books” but also “bookish,” “handbook,” and “booky.”

```
book 1
books 2
bookish, 3
handbook, 4
booky. 5
book? 6
book, 7
book! 8
Xbook! 9
!{book}! 10
```

```
whovian@phy504:~/host$ egrep '["\[{(!]*book.? ' sample_text_egrep3.txt
book 1
books 2
handbook, 4
book? 6
book, 7
book! 8
Xbook! 9
whovian@phy504:~/host$ egrep 'book[]})?!.,;:\\"s]* ' sample_text_egrep3.txt
book 1
books 2
handbook, 4
book? 6
book, 7
book! 8
Xbook! 9
!{book}! 10
```

# Regular Expressions

It is sometimes difficult to match a complete word. If we want to match the pattern “book”, our search will hit the word “book” and “books” but also “bookish,” “handbook,” and “booky.”

```
whovian@phy504:~/host$ egrep '["[{}(!]*book[]])?!.,;:"s]* ' sample_text_egrep3.txt
book 1
books 2
handbook, 4
book? 6
book, 7
book! 8
Xbook! 9
!{book}! 10
whovian@phy504:~/host$ egrep '["[{}(!]*book[]])?!.,;:"s]* ' sample_text_egrep3.txt
whovian@phy504:~/host$ █
```

Almost!

```
whovian@phy504:~/host$ egrep '(^ | )[\"[{}(!]*book[]])?!.,;:"s]* ' sample_text_egrep3.txt
book 1
books 2
book? 6
book, 7
book! 8
!{book}! 10
```

↑  
^ or space

Finally!

# regex + sed -E = power

In **sed**, we can (**regex**) to save the result of regex.

Then we can use \1, \2... to make replacements.

This is so powerful!

e.g, **shuf -i 1-10** print permutation without replacement

```
whovian@phy504:~/host$ shuf -i 1-10
7
10
9
3
8
1
5
2
4
6
```

# regex + sed -E = power

In `sed -E`, we can (regex) to save the result of regex.

Then we can use `\1, \2...` to make replacements.

e.g, `shuf -i 1-10` print permutation without replacement

Let's add a minus sign to all output

```
whovian@phy504:~/host$ shuf -i 1-10 | sed -E 's/([0-9]+)/-\1/'  
-3  
-1  
-6  
-2  
-10  
-8  
-7  
-9  
-5  
-4
```

The result of the regex [0-9]+ is saved on variable \1



# regex + sed -E = power

```
whovian@phy504:~/host$ shuf -i 1-10 | egrep "^[0-9]$"
```

```
7  
8  
3  
6  
4  
5  
1  
9  
2
```

Challenge: what if we add to a minus sign only to single digits numbers?

```
whovian@phy504:~/host$ shuf -i 1-10 | sed -E 's/([0-9])/-\1/'
```

```
-5  
-1  
-6  
-9  
-8  
-4  
-2  
-7  
-3  
10
```

We need to add the end of line regex \$ after [0-9] (one digit)

# regex + sed -E = power

```
whovian@phy504:~/host/Phy504Docker$ sed -E 's/(^.{1,80}).*/\1/' address_list.txt
John Daggett, 341 King Road, Plymouth MA
Alice Ford, 22 East Broadway, Richmond VA
Orville Thomas, 11345 Oak Bridge Road, Tulsa OK
Terry Kalkas, 402 Lans Road, Beaver Falls PA
Eric Adams, 20 Post Road, Sudbury MA
Hubert Sims, 328A Brook Road, Roanoke VA
Amy Wilde, 334 Bayshore Pkwy, Mountain View CA
Sal Carpenter, 73 6th Street, Boston MA
```

```
whovian@phy504:~/host/Phy504Docker$ sed -E 's/(^.{1,10}).*/\1/' address_list.txt
John Dagge
Alice Ford
Orville Th
Terry Kalk
Eric Adams
Hubert Sim
Amy Wilde,
Sal Carpen
```



Line truncation

# regex + sed -E = power

## Remove all alphanumeric

```
whovian@phy504:~/host/Phy504Docker$ more test_sed_regex.txt
cat stdlist
#ID      #Name
[101]    -Ali
[102]    -Neha
whovian@phy504:~/host/Phy504Docker$ sed 's/[A-Za-z0-9]//g' test_sed_regex.txt

#      #
[]    -
[]    -
```

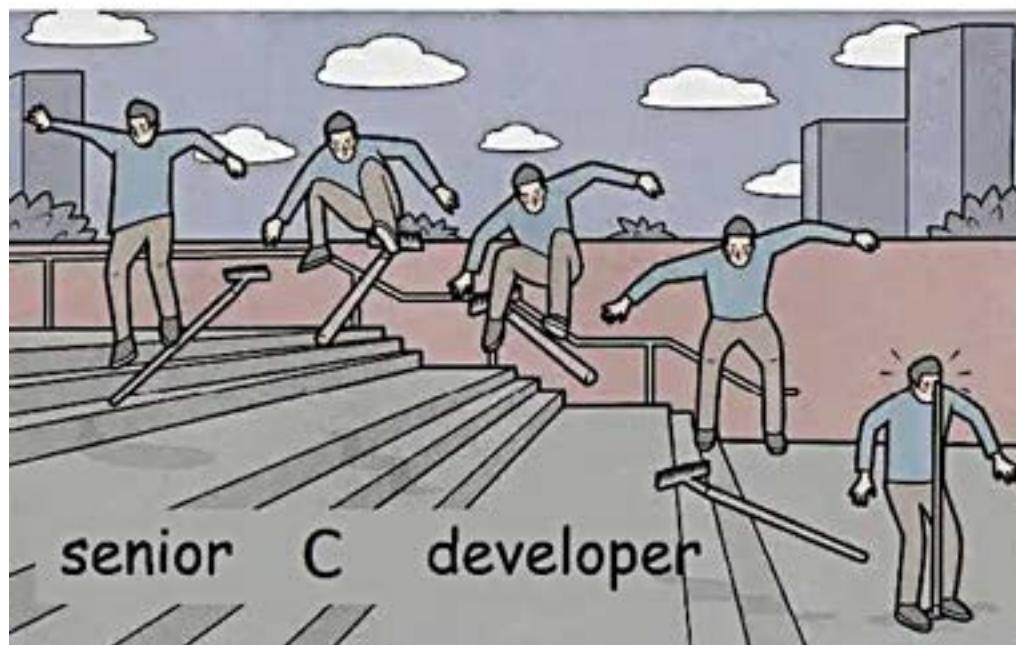
Replaces 4 consecutive space characters with tab character

```
whovian@phy504:~/host/Phy504Docker$ more test_sed_regex2.txt
1    2    3
4    5    6
7    9    10
whovian@phy504:~/host/Phy504Docker$ sed -e 's/    /\t/g' test_sed_regex2.txt
1        2        3
4        5        6
7        9        10
```

# Lecture 10: C part I

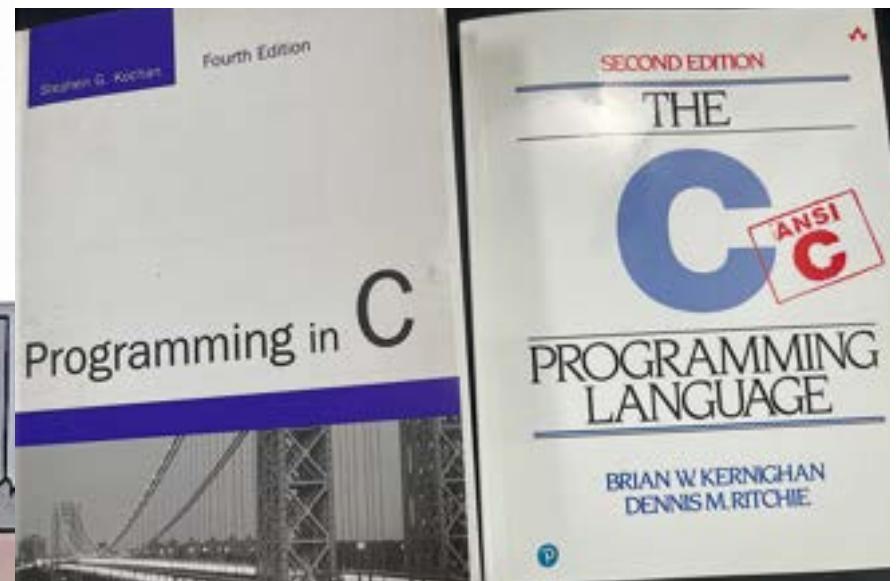


junior C developer



senior C developer

Suggested Literature



Suggested Literature

Both books are available at  
<https://learning.oreilly.com>

# C is a statically typed compiled language

- To run C/C++/Fortran you need a compiler. Example: the GNU compiler

```
whovian@phy504:~$ gcc --version  
gcc (Ubuntu 10.3.0-1ubuntu1~20.04) 10.3.0
```

**C compiler**

```
Copyright (C) 2020 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
whovian@phy504:~$ g++ --version  
g++ (Ubuntu 10.3.0-1ubuntu1~20.04) 10.3.0
```

**C++ compiler**

```
Copyright (C) 2020 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
whovian@phy504:~$ gfortran --version
```

**Fortran compiler**

```
GNU Fortran (Ubuntu 10.3.0-1ubuntu1~20.04) 10.3.0
```

```
Copyright (C) 2020 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

# Hello World in C

Hello World in C has many new components in comparison to bash

GNU nano 4.8

helloworld.c

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

Standard Library

```
int main (void)
{
```

All C programs need a main function

```
    printf ("Physics 504 is fun.\n");
```

Return value can be used by bash to check if the program run successfully

```
    return 0;
```

```
}
```

C requires ;

at the end of every line

# Hello World in C

Hello World in C has many new components in comparison to bash

GNU nano 4.8

helloworld.c

```
#include <stdio.h>
int main (void)
{
    printf ("Physics 504 is fun.\n");
    return 0;
}
```

Arguments of main allow command line arguments to be passed to C

printf here is similar in many ways to Bash

# Hello World in C

When you have only one file: one command compile and link the code (we will learn about Makefiles later for n files)

```
whovian@phy504:~/host/C$ nano helloworld.c
whovian@phy504:~/host/C$ gcc helloworld.c -o hello
whovian@phy504:~/host/C$ ./hello
Physics 504 is fun.
```

Compiler accepts many options that can perform conservative/aggressive optimizations, display warnings, or saves extra information that is useful for debuggers

```
whovian@phy504:~/host/C$ gcc -g3 -O0 helloworld.c -o hello
whovian@phy504:~/host/C$ ./hello
Physics 504 is fun.
```

provides information useful  
for debuggers (e.g., GDB)

disable float-point  
optimizations (to enable O1-3)

# Comments and variables in C

Two commands for comments in C `//` and `/* ... */`

```
GNU nano 4.8 sumint.c
#include <stdio.h> whovian@phy504:~/host/C$ gcc sumint.c -o sum
/*
Multi line observations whovian@phy504:~/host/C$ ./sum
Line 2!!
*/
The sum of 50 and 25 is 75
```

comments

```
int main (void)
{
    int value1, value2;

    value1 = 50;
    value2 = 25;
    int sum = value1 + value2; // you can declare inline

    printf ("The sum of %i and %i is %i\n", value1, value2, sum);

    return 0;
}
```

# int / float / double in C

GNU nano 4.8

```
#include <stdio.h>

int main (void)
{
    int      integerVar = 100;
    float    floatingVar = 331.79;
    double   doubleVar = 8.44e+11;
    char     charVar = 'W';
    _Bool    boolVar = 0;
```

```
printf ("integerVar = %i\n", integerVar);
printf ("floatingVar = %f\n", floatingVar);
printf ("doubleVar = %e\n", doubleVar);
printf ("doubleVar = %g\n", doubleVar);
printf ("charVar = %c\n", charVar);
printf ("boolVar = %i\n", boolVar);

return 0;
```

dtypes.c

```
whovian@phy504:~/host/C$ gcc dtypes.c -o dtypes
whovian@phy504:~/host/C$ ./dtypes
integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = W
boolVar = 0
```

Interesting discussion  
about Bool



# Integer variables in C

Be careful with integer division

```
GNU nano 4.8      intdiv.c      Modified  
#include <stdio.h>  
  
int main (void)  
{  
    int a = 1;  
    int b = 2;  
    int c = a/b;  
  
    double d = a/b;  
    double e = a / (double) b;  
    double f = 1/2;  
    double g = 1./2.;  
  
    printf ("c = %i\n", c); whovian@phy504:~/host/C$ gcc intdiv.c -o idiv  
    printf ("d = %f\n", d); whovian@phy504:~/host/C$ ./idiv  
    printf ("e = %f\n", e); c = 0  
    printf ("f = %f\n", f); d = 0.000000  
    printf ("g = %f\n", g); e = 0.500000  
    return 0;  
}
```

casting



```
GNU nano 4.8          floatprec.c      Mo
#include <stdio.h>

int main (void)
{
    // float has 10^-7 prec. max
    float a = 1.0012033;
    float b = 1.0012034;
    float c = b - a;

    printf ("c = %g\n", c);

    double f = 1.0012033;
    double g = 1.0012034;
    double h = g - f;

    printf ("h = %g\n", h);

    return 0;
}
```

# float/double variables in C

[Good discussion on StackOverflow](#) on **float** vs **double** vs **long double**

[What Every Computer Scientist Should Know About Floating-Point Arithmetic](#)

```
whovian@phy504:~/host/C$ nano floatprec.c
whovian@phy504:~/host/C$ gcc floatprec.c
whovian@phy504:~/host/C$ ./float
c = 1.19209e-07
h = 1e-07
whovian@phy504:~/host/C$
```



wrong result

# Be careful: casting is not rounding

```
GNU nano 4.8          castvsround.c
#include <stdio.h>
#include <math.h>

int main (void)
{
    double a = 2.9;
    int b = (int) a;
    int c = round(a);

    printf ("b = %i\n", b);
    printf ("c = %i\n", c);

    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc castvsround.c -o castvsround
/usr/bin/ld: /tmp/cc8BHsBG.o: in function `main':
castvsround.c:(.text+0x2f): undefined reference to `round'
collect2: error: ld returned 1 exit status

whovian@phy504:~/host/C$ g++ castvsround.c -o castvsround
whovian@phy504:~/host/C$
```

We must link the math library manually  
(historical reasons)  
(btw: not in C++)

```
whovian@phy504:~/host/C$ gcc castvsround.c -o castvsround -lm
whovian@phy504:~/host/C$ ./castvsround
b = 2
c = 3
```

GNU nano 4.8

overflow.c

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

int main()
{
    // INT_MAX is the maximum
    // representable integer.
    int a = INT_MAX;

    printf("a = %i\n", a);

    a = a + 1;

    printf("a = %i\n", a);

    long int b = ((long int) INT_MAX)*2;

    printf("b = %li\n", b);

    return 0;
}
```

Be careful:  
under/overflow  
(e.g., **int** vs **long int**)

Wrong result  
(overflow)

first cast to long then  
multiply [Link on  
Under/Overflow](#)

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

int main()
{
    // INT_MAX is the maximum
    // representable integer.
    unsigned int a = INT_MAX;

    printf("a = %i\n", a);

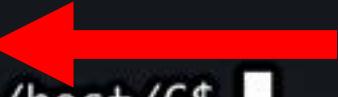
    a = a + 1; 
    printf("a = %u\n", a);

    return 0;
}
```

## Signed vs unsigned int

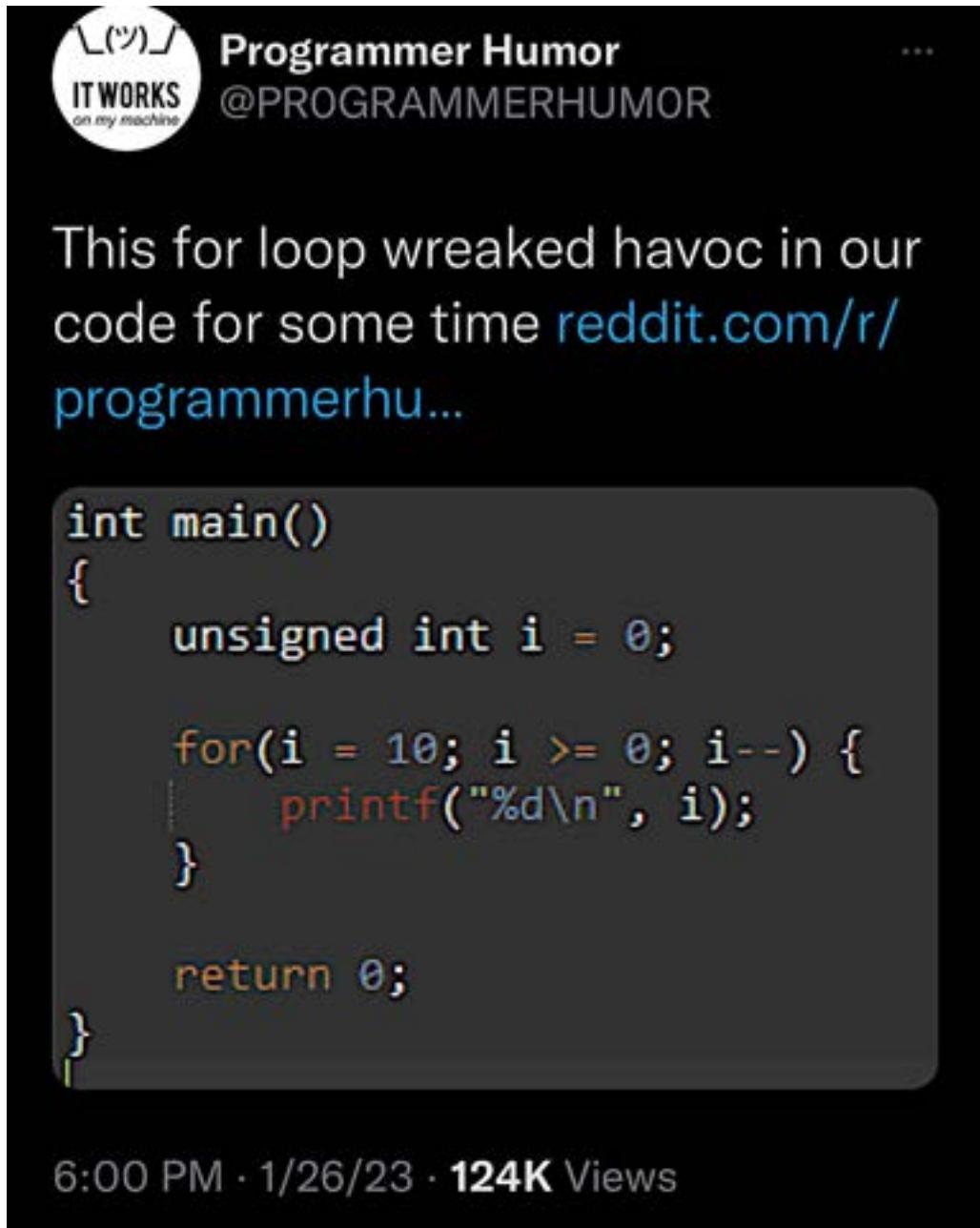
[Good discussion on StackOverflow](#)

Not having to carry a bit for integer sign double the range without overflow

 correct result

```
whovian@phy504:~/host/C$ gcc overflow.c -o overflow
whovian@phy504:~/host/C$ ./overflow
a = 2147483647
a = 2147483648
whovian@phy504:~/host/C$ █
```

# Be careful with unsigned int



A screenshot of a Reddit post from the subreddit /r/ProgrammerHumor. The post has a white circular profile picture with the text "IT WORKS on my machine" and a sad face icon. The title of the post is "Be careful with unsigned int". Below the title, it says "Programmer Humor" and "@PROGRAMMERHUMOR". There is a "... more" link. The post content is a C code snippet:

```
int main()
{
    unsigned int i = 0;

    for(i = 10; i >= 0; i--) {
        printf("%d\n", i);
    }

    return 0;
}
```

At the bottom of the post, it shows "6:00 PM · 1/26/23 · 124K Views".

This loop runs forever (try yourself)!

There is no negative sign w/ **unsigned int**

When doing **i--** on **i=0** iteration, the variable underflows (assigned garbage, but positive garbage!)

# Complex Numbers

```
GNU nano 4.8                         complex.c
#include <stdio.h>
#include <complex.h>

int main()
{
    double complex z1 = 1.0 + 3.0 * I;
    double complex z2 = 1.0 - 4.0 * I;

    double complex a = z1 + z2;
    printf("sum: Z1+Z2 = %f + %fi \n", creal(a), cimag(a));

    double complex b = z1 - z2;
    printf("diff: Z1-Z2 = %f + %fi \n", creal(b), cimag(b));

    return 0;
}
```



There is no %something for complex numbers in C

# Assignment Operators

```
#include <stdio.h>

int main (void)
{
    int a = 10;
    printf ("a = %i\n", a);
    a += 10;
    printf ("a = %i\n", a);
    a *= 8;
    printf ("a = %i\n", a);
    a /= 4;
    printf ("a = %i\n", a);

    return 0;
}
```

C allows joint arithmetic operators with assignment

$a = a + 10$

$a = a * 8$

$a = a / 4$

```
whovian@phy504:~/host/C$ nano aoper.c
whovian@phy504:~/host/C$ gcc aoper.c -o aoper
whovian@phy504:~/host/C$ ./aoper
a = 10
a = 20
a = 160
a = 40
```

# Relational operators

Table 4.1 Relational Operators

Operator	Meaning	Example
<code>==</code>	Equal to	<code>count == 10</code>
<code>!=</code>	Not equal to	<code>flag != DONE</code>
<code>&lt;</code>	Less than	<code>a &lt; b</code>
<code>&lt;=</code>	Less than or equal to	<code>low &lt;= high</code>
<code>&gt;</code>	Greater than	<code>pointer &gt; endOfList</code>
<code>&gt;=</code>	Greater than or equal to	<code>j &gt;= 0</code>

Important point: the relational operators have lower precedence than all arithmetic operators.

$$a < b + c$$

First  $b + c$  will be evaluated, then compared, i.e., expression above is the equivalent to  $a < (b + c)$

PS:  $a <= 200$  is equivalent to  $a < 201$ . I prefer "less than" or "greater than." Why? Equality on float numbers is complicated. So, I view it as good practice to take `<=` and `=>` operators out of your mind when coding (even when dealing with integers)

# EXIT CODES

`stdlib.h` contains **EXIT\_SUCCESS** and **EXIT\_FAILURE** macros expand into integral expressions that can be used as arguments to the exit function

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

int main()
{
    printf("Hello World\n");
    return EXIT_SUCCESS;
}
```

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

int main()
{
    printf("Ops, Failure\n");
    return EXIT_FAILURE;
}
```

```
whovian@phy504:~/host$ gcc exitsuccess.c -o exitsuccess
whovian@phy504:~/host$ ./exitsuccess
Hello World
whovian@phy504:~/host$ echo $?
0
```

```
whovian@phy504:~/host$ gcc exitfailure.c -o exitfailure
whovian@phy504:~/host$ ./exitfailure
Ops, Failure
whovian@phy504:~/host$ echo $?
1
```

good coding practice to use **EXIT\_SUCCESS** and **EXIT\_FAILURE**

# Compiler is your friend

You can ask the compiler to  
be extra careful

Compiler warnings can be  
life saving

You should aim to  
resolve/understand all  
warnings from your code

My compiler watching me continue to  
code, ignoring its warnings:



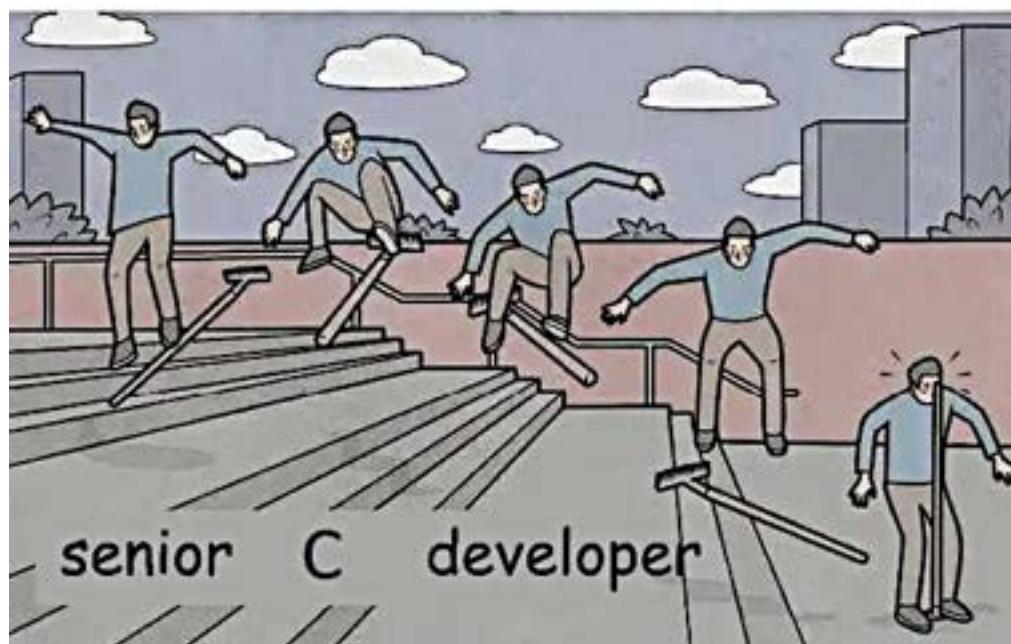
The things my compiler sees..

- **-Wall** — There is high confidence in both the value and low false-positive rate of these warnings
- **-Wextra** — These warnings are believed to be valuable and robust (not buggy), but they may have high false-positive rates or common philosophical objections
- **-Wpedantic** — Enforces strict ISO C/C++.

# Lecture 11: C part II

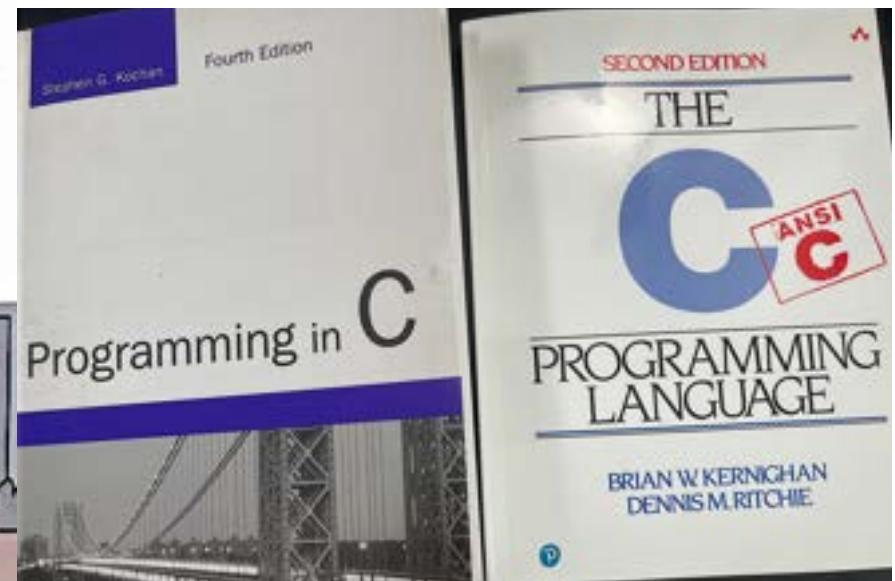


junior C developer



senior C developer

## Suggested Literature

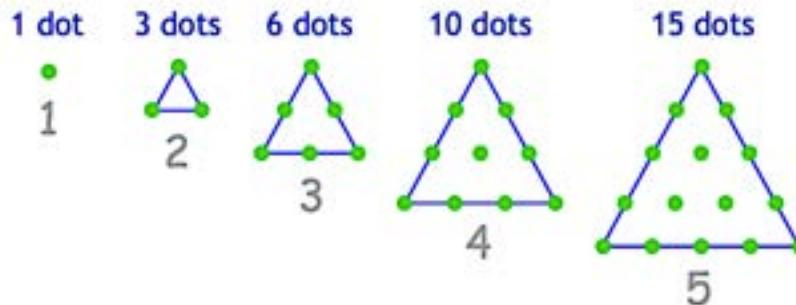


## Suggested Literature

Both books are available at  
<https://learning.oreilly.com>

# Loops (**for**)

Triangular number sequence: 1, 3, 6, 10, 15, 21, 28, 36, 45, ...



This algorithm is simple to be implemented in a loop

By adding another row of dots and counting all the dots we can find the next number of the sequence.

GNU nano 4.8  
#include <stdio.h>

```
whovian@phy504:~/host$ gcc trinumloop.c -o trinumloop
whovian@phy504:~/host$ ./trinumloop
The eighth triangular number is 36
```

```
int main ()
{
    // compute the 8th triangular number
    int x = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8;

    printf ("The eighth triangular number is %i\n",  x);

    return 0;
}
```

C code without a loop is unfeasible for large values  
PS: ok to declare and define in the same variable

GNU nano 4.8

trinumloop.c

```
// Program to calculate the 200th triangular number
#include <stdio.h>

int main (void)
{
    int x = 0; // SUPER IMPORTANT to initialize the variable
               // with zero EXPLICITLY
    for (int n = 1; n <= 200; n = n + 1)
    {
        // PS: you don't need {} for one-line for loops. However
        // I believe it is good practice to always include them
        x += n;
    }
    printf ("The 200th triangular number is %i\n", x);

    // The issue of initialization
    int y = 0;
    int z; // This will have random garbage! Sometimes by chance z
           // will be zero (this is what we called undefined behavior
    printf ("y and z = %i and %i\n", y, z);

    return 0;
}
```

whovian@phy504:~/host\$ gcc trinumloop.c -o trinumloop  
whovian@phy504:~/host\$ ./trinumloop  
The 200th triangular number is 20100  
y and z = 0 and 0

don't let this fool you

Good practice for multicore to init variable **n** in the loop def

Can this loop be threaded (multicore)? NO

# Loops (**for**)

```
GNU nano 4.8
#include <stdio.h>

int main (void)
{ // Generate a table of triangular numbers
    printf ("TABLE OF TRIANGULAR NUMBERS\n\n");
    printf (" n      Sum from 1 to n\n");
    printf ("---  -----\n");

    int tn = 0;          I am using short name vars for
    for (int n = 1;  n < 11;  ++n)   display (but NOT ok in general!)
    {
        tn += n;
        printf ("%2i      %i\n", n, tn);
    }
    return 0;
}
```

A red arrow points upwards from the line `%2i` to the line `tn += n;`. Another red arrow points to the `++n` part of the `for` loop's increment expression.

**This makes sure there is no shift between 9 and 10 (try without it)**

Future: can't parallelize loops with **printf**

TABLE OF TRIANGULAR NUMBERS	
n	Sum from 1 to n
---	-----
1	1
2	3
3	6
4	10
5	15
6	21
7	28
8	36
9	45
10	55

# Loops (**for**) with a touch with scanf scan for input

```
#include <stdio.h>

int main (void)
{
    int m;
    printf ("What triangular number do you want? ");
    scanf ("%i", &m);
    int tn = 0;
    for (int n = 1; n < m+1; ++n)
    {
        tn += n;
    }
    printf ("Triangular number %i is %i\n", m, tn);
    return 0;
}
```

```
whovian@phy504:~/host$ gcc printtrinumloop2.c -o printtrinumloop2
whovian@phy504:~/host$ ./printtrinumloop2
What triangular number do you want? 10
Triangular number 10 is 55
```

Reminder: don't use short cryptic names

(I am using here to fit code in a slide)

&x is the **reference** to variable x  
(basically its memory position)  
so scanf knows where to write it

Reminder: many books don't do that, but  
you can (and should) define variables as  
local as possible (here inside the loop)

GNU nano 4.8

nestedprintnumloop.c

Modified

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

```
int main (void)
```

```
{
```

```
    for (int c=1; c<6; ++c)
```

```
{
```

```
        int m;
```



Be local: if you only need the variable  
inside the loop, declare it inside the loop!

```
        printf ("What triangular number do you want?");
```

```
        scanf ("%i", &m);
```

```
        int tn = 0;
```



Be local: I only need **tn** below, I

```
        for (int n=1; n<m+1; ++n) only declare tn here
```

```
{
```

```
        tn += n;
```

```
}
```

```
        printf ("Triangular number %i is %i\n\n", m, tn);
```

```
}
```

```
return 0;
```

**RUN THIS PROGRAM**

# Loops (**while**)

```
#include <stdio.h> whovian@phy504:~/host$ gcc whileloops.c -o whileloop  
whovian@phy504:~/host$ ./whileloops  
int main (void)  
{  
    int ct = 1;  
    First Run: 1  
    First Run: 2  
    First Run: 3  
    First Run: 4  
    First Run: 5  
    while (ct < 6)  
    {  
        printf ("First Run: %i\n", ct);  
        ++ct;  
    }  
    // while may never run!  
    while (ct < 6) ←  
    {  
        printf ("Second run: %i\n", ct);  
        ++ct;  
    }  
    return 0;  
}
```

If the condition fail right away, the program skip over the code inside

# Loops (**while**) – interesting case

```
#include <stdio.h>
int main()
{
    printf( "\t--- Calculate Averages ---\n" );
    printf( "Enter some small numbers (type a letter to end input) \n" );

    double x;
    double sum = 0;
    int count = 0;

    while (scanf("%lf", &x) == 1)
    {
        printf("x = %lf\n", x);
        sum += x;
        ++count;
    }
    if (count == 0)
    {
        printf("No input data!\n");
    }
    else
    {
        printf("Average = %.2f\n", sum/count );
    }
    return 0;
}
```

scanf does return an integer!  
scanf returns the number of fields that were successfully converted and assigned

# Loops (**while**) – interesting case

```
whovian@phy504:~/host$ ./whileloops2
    --- Calculate Averages ---
Enter some small numbers (type a letter to end input)
1 2 3 4
x = 1.000000
x = 2.000000
x = 3.000000
x = 4.000000
5678
x = 5678.000000
2 4 5
x = 2.000000
x = 4.000000
x = 5.000000
X
Average = 712.38
```

The while loop produces an interesting behavior. As long as you write numbers, the program keeps waiting for more input

# Loops (do-while)

```
#include <stdio.h>
int main (void)
{
    int ct = 1;
    do
    {
        printf ("First Run: %i\n", ct);
        ++ct;
    } while (ct < 6);
    do
    {
        printf ("Second run: %i\n", ct);
        ++ct;
    } while (ct < 6);
    return 0;
}
```

```
whovian@phy504:~/host$ gcc dowhileloops.c -o dowhileloops
whovian@phy504:~/host$ ./dowhileloops
First Run: 1
First Run: 2
First Run: 3
First Run: 4
First Run: 5
Second run: 6
```

we need semicolon here

Do loops always run at least once!

```
GNU nano 4.8          break.c      Modified  
#include <stdio.h>  
  
int main (void)  
{  
    for (int n=0;  n<5;  ++n)  
    {  
        if (n==2)      break will exit the  
        {              loop completely  
            break;  
        }  
        printf ("Testing Break  %i\n", n);  
    }  
    for (int n=0;  n<5;  ++n)  
    {  
        if (n==2)      continue will skip one  
        {              iteration of the loop  
            continue;  
        }  
        printf ("Testing Continue  %i\n", n);  
    }  
    return 0;  
}
```

# break / continue statements

```
whovian@phy504:~/hos:  
whovian@phy504:~/hos:  
Testing Break  0  
Testing Break  1  
Testing Continue  0  
Testing Continue  1  
Testing Continue  3  
Testing Continue  4
```

```
GNU nano 4.8                                break2.c
#include <stdio.h>

int main (void)
{
    for (int n=0;  n<3;  ++n)
    {
        for (int m=0;  m<3;  ++m)
        {
            if (m==1)
            {
                break;
            }
            printf ("Testing Break n=%i m=%i\n", n, m);
        }
    }
    for (int n=0;  n<3;  ++n)
    {
        for (int m=0;  m<3;  ++m)
        {
            if (m==1)
            {
                continue;
            }
            printf ("Testing Continue n=%i m=%i\n", n, m);
        }
    }
    return 0;
}
```

break will exit only  
the inner loop

continue will only  
skip one iteration of  
the inner loop

# break / continue statements

```
whovian@phy504:~/host$ ./
Testing Break n=0 m=0
Testing Break n=1 m=0
Testing Break n=2 m=0
Testing Continue n=0 m=0
Testing Continue n=0 m=2
Testing Continue n=1 m=0
Testing Continue n=1 m=2
Testing Continue n=2 m=0
Testing Continue n=2 m=2
```

# if / else statements

```
#include <stdio.h>

int main (void)
{
    int num;
    printf ("Type in your number: ");
    scanf ("%i", &num);

    if (num < 0)      You don't need to have the
    {                  else part if not needed
        num = -num;
    }

    printf ("The absolute value is %i\n", num);
    return 0;
}
```

# if / else statements

```
#include <stdio.h>

int main ()
{
    int num;
    printf ("Enter your number to be tested: ");
    scanf ("%i", &num);

    int rem = num % 2;

    if (rem == 0)
    {
        printf ("The number is even.\n");
    }
    else
    {
        printf ("The number is odd.\n");
    }
    return 0;
}
```

```
whovian@phy504:~/host$ ./ifelse2
Enter your number to be tested: 4
The number is even.
whovian@phy504:~/host$ ./ifelse2
Enter your number to be tested: 5
The number is odd.
```

# if / else statement in one line (quite useful)

```
#include <stdio.h>

int main ()
{
    int num;
    printf ("Enter your number to be tested: ");
    scanf ("%i", &num);

    const int rem = (num % 2) == 0 ? 0 : 1;

    if (rem == 0)
    {
        printf ("The number is even.\n");
    }
    else if (rem == 1)
    {
        printf ("The number is odd.\n");
    }
    else
    {
        printf("This should be impossible\n");
    }
    return 0;
}
```

This is the only way to use **if/else** to define constant variables

This code also illustrates how to write code with **if / else if / else**

# if / else statement in one-line (useful!)

```
GNU nano 4.8          ifelse5.c
#include <stdio.h>

int main ()
{
    int num;
    printf ("Enter the year to be tested: ");
    scanf ("%i", &num);

    const int res = (num < 0) ? -1 : (num == 0) ? 0 : 1;

    printf ("The sign of the number is %i \n", res);

    return 0;
}
```

You can have nested on-line conditional statements ( ?: )

```
whovian@phy504:~/host$ ./ifelse5
Enter the year to be tested: 200
The sign of the number is 1
whovian@phy504:~/host$ ./ifelse5
Enter the year to be tested: 0
The sign of the number is 0
whovian@phy504:~/host$ ./ifelse5
Enter the year to be tested: -100
The sign of the number is -1
```

# if / else statement with logical operators

```
#include <stdio.h>

int main ()
{
    int num;
    printf ("Enter the year to be tested: ");
    scanf ("%i", &num);

    const int rem4 = num % 4;
    const int rem100 = num % 100;
    const int rem400 = num % 400;

    if ((rem4 == 0  &&  rem100 != 0) || rem400 == 0)
    {
        printf ("It is a leap year.\n");
    }
    else
    {
        printf ("It is not a leap year.\n");
    }
    return 0;
}
```

General advice: if a variable is not going to be modified, make it **const!**

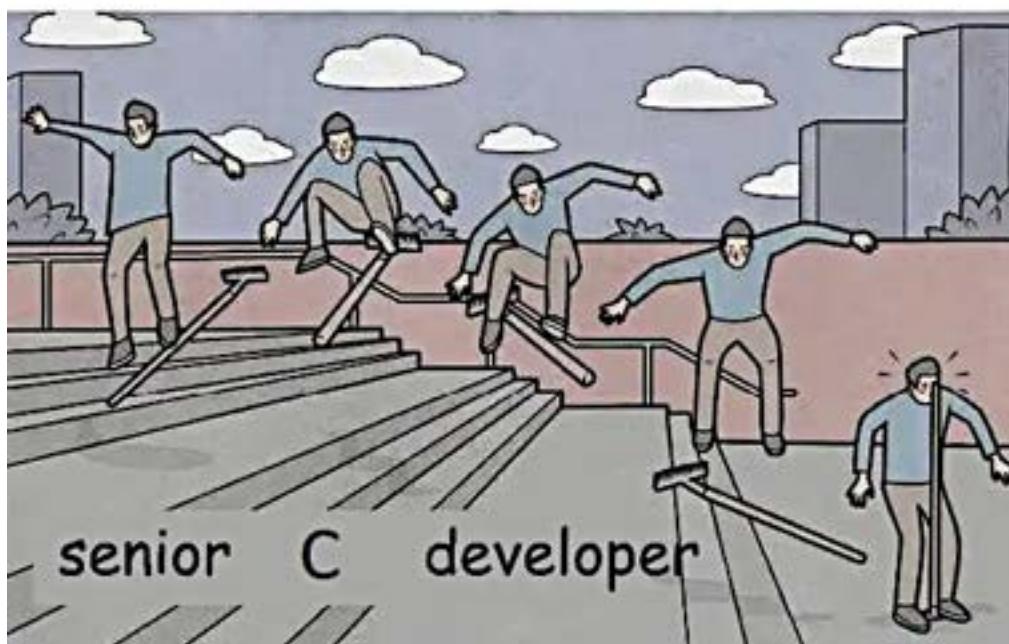
That helps debugging and optimization

```
whovian@phy504:~/host$ gcc ifelse4.c -o ifelse4
whovian@phy504:~/host$ ./ifelse4
Enter the year to be tested: 1986
It is not a leap year.
```

# Lecture 12: C part III

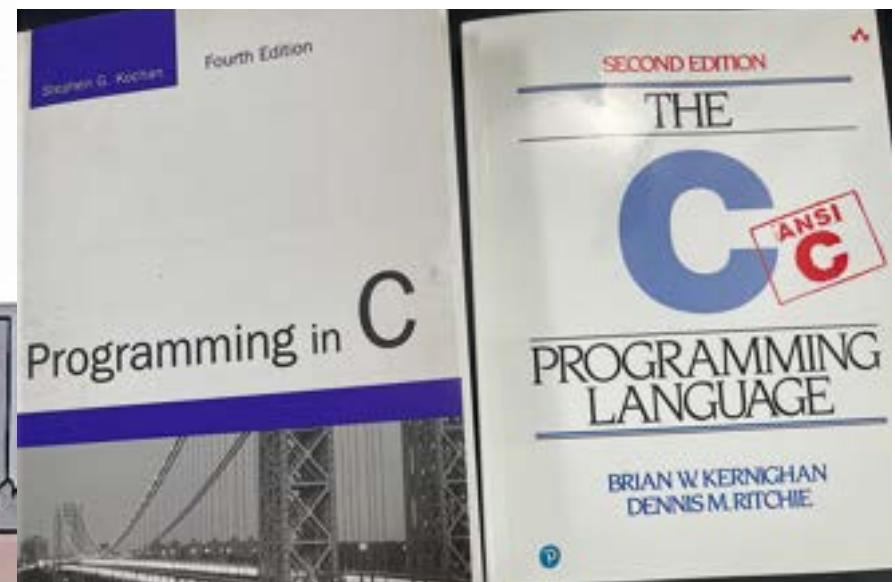


junior C developer



senior C developer

## Suggested Literature



## Suggested Literature

Both books are available at  
<https://learning.oreilly.com>

# Arrays

There are two ways to define arrays in C. Today we will learn one of them

GNU nano 4.8

```
#include <stdio.h>

int main (void)
{
    int x[5];
    x[0] = 197;
    x[1] = 200;
    x[2] = -100;
    x[3] = x[0] + x[1];
    x[4] = 350;

    for (int i=0; i<5; ++i)
    {
        printf ("x[%i] = %i\n", i, x[i]);
    }
    return 0;
}
```

size of the array

counting start from zero

```
whovian@phy504:~/host$ ./arrays
x[0] = 197
x[1] = 200
x[2] = -100
x[3] = 397
x[4] = 350
```

# Arrays

Array **slicing** goes from zero to size - 1

```
#include <stdio.h>

int main (void)
{
    const int size = 5;
    int x[size];

    x[0] = 197;           < size (this means last
    x[1] = 200;           element is size-1)
    x[2] = -100;
    x[3] = x[0] + x[1];
    x[4] = 350;

    for (int i=0; i<size; ++i)
    {   // in C - arrays goes from zero to size - 1
        printf ("x[%i] = %i\n", i, x[i]);
    }
    return 0;
}
```

# Arrays

The size of the array can be chosen at runtime as well!

```
#include <stdio.h>

int main (void)
{
    int size;
    scanf ("%i", &size);
    int x[size];
    for (int i=0; i<size; ++i)
    {
        // in C - arrays goes from zero to size - 1
        x[i] = 2* i;
        printf ("x[%i] = %i\n", i, x[i]);
    }
    return 0;
}
```

compiler does not know the  
size of the array!

This is called  
**Variable Length  
Arrays** (caution: optional  
feature in C11)

```
whovian@phy
whovian@phy
10
x[0] = 0
x[1] = 2
x[2] = 4
x[3] = 6
x[4] = 8
x[5] = 10
x[6] = 12
x[7] = 14
x[8] = 16
x[9] = 18
```

[Be careful with large arrays](#) (link from StackOverflow about StackOverflow!)

# Arrays

## Alternative way to initialize arrays

```
#include <stdio.h>

int main (void)
{
    int x[10] = {0, 1, 4, 9, 16}; 
    for (int i=5; i<10; ++i)
    {
        x[i] = i * i;
    }

    for (int i=0; i<10; i++)
    {
        printf ("x[%i] = %i\n", i, x[i]);
    }

    return 0;
}
```

```
whovian@ph
whovian@ph
x[0] = 0
x[1] = 1
x[2] = 4
x[3] = 9
x[4] = 16
x[5] = 25
x[6] = 36
x[7] = 49
x[8] = 64
x[9] = 81
```

# Multi-dimensional Arrays

You can create n-dimensional arrays easily

```
#include <stdio.h>

int main (void)
{
    int x[4][5] = {{ 10,  5, -3, 17, 82 },
                   {  9,  0,  0,  8, -7 },
                   { 32, 20,  1,  0, 14 },
                   {  0,  0,  8,  7,  6 }};

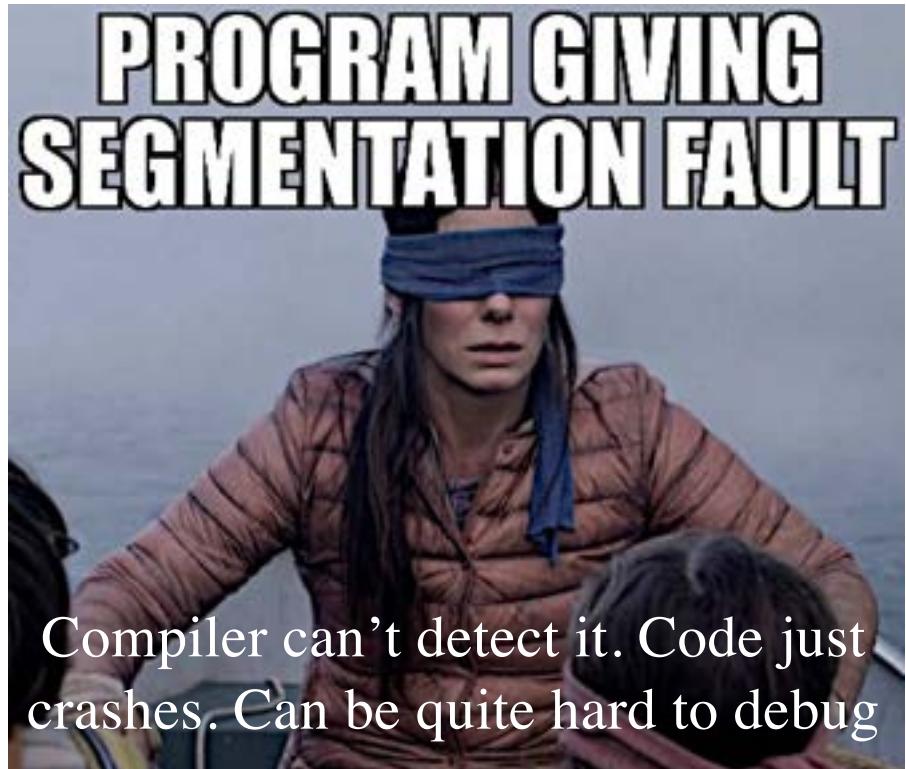
    for (int i=0; i<4; i++)
    {
        for (int j=0; j<5; j++)
        {
            printf ("i = %i, j = %i, a_ij = %i\n",
                   i, j, x[i][j]);
        }
    }

    return 0;
}
```

```
whovian@phy504:~/host$ ./array
i = 0, j = 0, a_ij = 10
i = 0, j = 1, a_ij = 5
i = 0, j = 2, a_ij = -3
i = 0, j = 3, a_ij = 17
i = 0, j = 4, a_ij = 82
i = 1, j = 0, a_ij = 9
i = 1, j = 1, a_ij = 0
i = 1, j = 2, a_ij = 0
i = 1, j = 3, a_ij = 8
i = 1, j = 4, a_ij = -7
i = 2, j = 0, a_ij = 32
i = 2, j = 1, a_ij = 20
i = 2, j = 2, a_ij = 1
i = 2, j = 3, a_ij = 0
i = 2, j = 4, a_ij = 14
i = 3, j = 0, a_ij = 0
i = 3, j = 1, a_ij = 0
i = 3, j = 2, a_ij = 8
i = 3, j = 3, a_ij = 7
i = 3, j = 4, a_ij = 6
```

# Arrays

You can write on array locations that have no memory allocation: **segfault**



- Ref. 1 from [codeforgeeks](#)
- Ref. 2 from [StackOverflow](#)
- Ref. 3 from [StackOverflow](#)

# Arrays

Even worse than segfault is when C just return garbage (out of bounds)

```
#include <stdio.h>

int main (void)
{
    int x[5];
    x[0] = 197;
    x[1] = 200;
    x[2] = -100;
    x[3] = x[0] + x[1];
    x[4] = 350;

    for (int i=0; i<100; ++i)
    {
        printf ("x[%i] = %i\n", i, x[i]);
    }
    return 0;
}
```

out of bounds access to  
array is **undefined  
behavior** (it could cause  
a segmentation fault, but  
it doesn't have to)

```
whovian@phy504:~/hos
whovian@phy504:~/hos
x[0] = 197
x[1] = 200
x[2] = -100
x[3] = 397
x[4] = 350
x[5] = 64
x[6] = 946356992
x[7] = 244184714
x[8] = 0
x[9] = 0
x[10] = 25616515
x[11] = 64
x[12] = 107
x[13] = 1
x[14] = 25187512
x[15] = 64
x[16] = 27469728
x[17] = 1
x[18] = 4457
```

# Arrays

Undefined behavior is the most difficult error to debug (the code just runs)



- Ref. 1 from [codeforgeeks](#)
- Ref. 2 from Prof. [John Regehr's Blog](#)

# Functions in C

```
GNU nano 4.8
#include <stdio.h>

void swap(int a, int b)
{
    const int t = a;
    a = b;
    b = t;

    printf("swap: a = %d, b = %d\n", a, b);
}

int main ()
{
    const int x = 2;
    const int y = 3;

    swap (2, 3);whovian@phy504:~/host$ gcc functions.c -o functions
    whovian@phy504:~/host$ ./functions
    swap: a = 3, b = 2
    whovian@phy504:~/host$ █
```

Functions accept arguments, do something, and return values

Of course, this is C, and any basic definitions can be tweaked.

Functions may not return anything

Functions may also not accept arguments

# Functions in C

```
#include <stdio.h>
void swap(int a, int b)
{
    const int t = a;
    a = b;
    b = t;

    printf("swap: a = %d, b = %d\n", a, b);
}

int main ()
{
    int x = 2;
    int y = 3;
    swap (x, y);

    printf("swap: a = %d, b = %d\n", x, y);

    return 0;
}
```

x and y are copied

x and y are copied

Function copy arguments (**pass by value**). You can't change the parameter values via arguments\*

\*Unless you use pointers (next class)

# Functions in C

Functions may not have any arguments

```
#include <stdio.h>

void PrintMessage (void)
{
    printf ("Programming is fun. But this"
            " function is boring.\n");
}

int main ()
{
    PrintMessage();

    return 0;
}
```

```
whovian@phy504:~/host$ gcc functions3.c -o functions3
whovian@phy504:~/host$ ./functions3
Programming is fun. But this function is boring.
whovian@phy504:~/host$ █
```

# Functions in C

## Function return values (just one!)

```
#include <stdio.h>
double KmToMph (const double kmh)
{
    return kmh * 0.62137119223;
}

int main ()
{
    printf("1 km/h corresponds to "
           "%lf miles/h \n", KmToMph(1));

    return 0;
}
```

Having “magic numbers” out of thin air is bad coding practice.

Btw: not enough digits to maintain double precision

```
whovian@phy504:~/host$ gcc functions4.c -o functions4
whovian@phy504:~/host$ ./functions4
1 km/h corresponds to 0.621371 miles/h
```

```

#include <stdio.h>

int min (int vals[10])
{
    int tmp = vals[0];

    for (int i=1; i<10; ++i)
    {
        if ( vals[i] < tmp )
        {
            tmp = vals[i];
        }
    }
    return tmp;
}

int main (void)
{
    int scores[10] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8};
    printf ("Minimum score is %i\n", min(scores));

    int scores2[5] = {2, 1, 3, 4, 5};
    printf ("Minimum score is %i\n", min(scores2));

    int scores3[11] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8, 0};
    printf ("Minimum score is %i\n", min(scores3));

    return 0;
}

```

# Functions in C

mildly dangerous  
 (but important case): arrays

```

whovian@phy504:~/host$ ./functions5
Minimum score is 1
Minimum score is 0 ← wrong
Minimum score is 1 ← wrong

```

## UNDEFINED BEHAVIOUR

May be right/wrong – who knows? Nobody! Undefined

Wrong – but behavior is predictable

```
#include <stdio.h>

int min (int n, int vals[])
{
    if (n <= 0) ← Check input
    {
        printf ("Error, size = %i \n", n);
    }

    int tmp = vals[0];

    for (int i=1; i<n; ++i)
    {
        if ( vals[i] < tmp )
        {
            tmp = vals[i];
        }
    }
    return tmp;
}

int main (void)
{
    int scores[10] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8};
    printf ("Minimum score is %i\n", min(10, scores));

    int scores2[5] = {2, 1, 3, 4, 5};
    printf ("Minimum score is %i\n", min(5, scores2));

    int scores3[11] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8, 0};
    printf ("Minimum score is %i\n", min(11, scores3));

    return 0;
}
```

# Functions in C

With VLAs and an extra integer argument, we can fix the problem

```
whovian@phy504:~/host$ ./functions5
Minimum score is 1
Minimum score is 1
Minimum score is 0
```



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

```
int min (int vals[10])
{
    int tmp = vals[0];

    for (int i=1; i<10; ++i)
    {
        if ( vals[i] < tmp )
        {
            tmp = vals[i];
        }
    }
    vals[1] = -1;
    return tmp;
}
```

```
int main (void)
```

```
{
    int scores[10] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8};
    printf ("Minimum score is %i\n", min(scores));
    printf ("scores[1] = %i \n", scores[1]);

    return 0;
}
```

# Functions in C

With arrays we CAN change the values of the function input

```
whovian@phy504:~/host$ ./functions7
Minimum score is 1
scores[1] = -1
```

Why? Because under the hood arrays are basically pointers!  
(see next class)

# Functions in C

```
#include <stdio.h>

int min (const int vals[10])
{
    int tmp = vals[0];

    for (int i=1; i<10; ++i)
    {
        if ( vals[i] < tmp )
        {
            tmp = vals[i];
        }
    }
    vals[1] = -1;
    return tmp;
}

int main (void)
{
    int scores[10] = {2, 1, 3, 4, 5,
                      6, 9, 10, 7, 8};
    printf ("Minimum score is %i\n", min(scores));
    printf ("scores[1] = %i \n", scores[1]);

    return 0;
}
```

```
whovian@phy504:~/host$ gcc functions8.c -o functions8
functions8.c: In function 'min':
functions8.c:14:13: error: assignment of read-only location '*(&vals + 4)'
  14 |         vals[1] = -1;
          ^
```

If your function does not change array values, make the argument constant!

It also helps the compiler

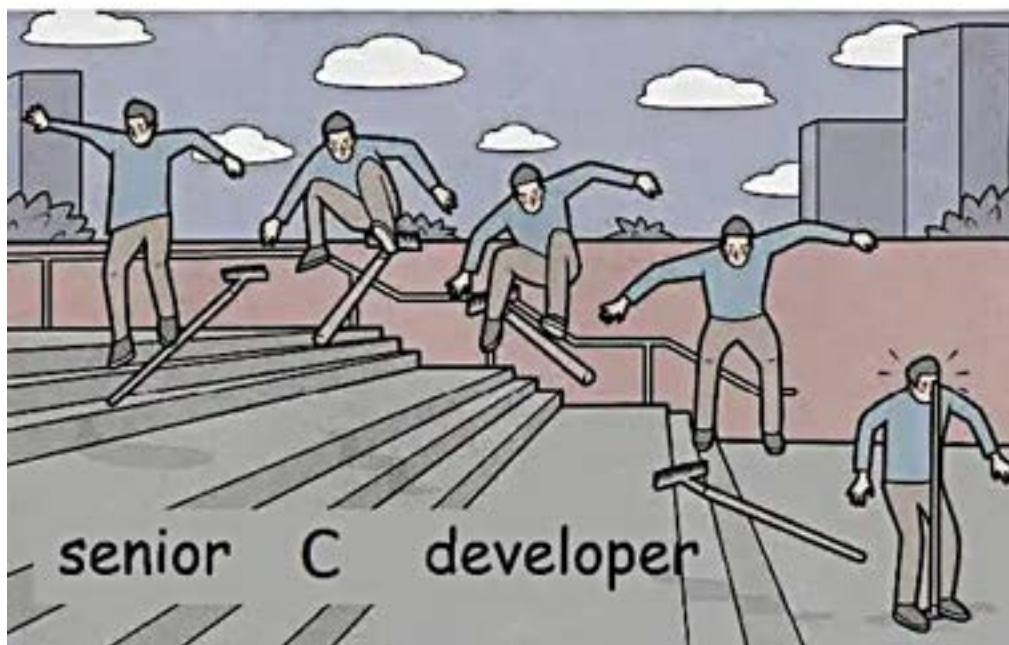
# Lecture 13: C part IV



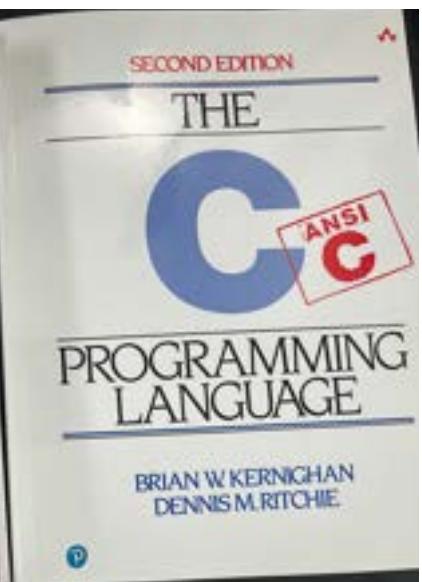
junior C developer



Suggested Literature



senior C developer



# Static variables (in functions)

```
#include <stdio.h>
int fun(void)
{
    static int count = 0;
    count++;
    return count;
}
int main(void)
{
    printf("count = %d \n", fun());
    return 0;
}
```

extremely  
dangerous in  
parallel computation

```
whovian@phy504:~/host$ gcc functions9.c -o functions9
whovian@phy504:~/host$ ./functions9
count = 1
count = 2
count = 3
count = 4
```

Variable remember  
values between calls

**Dangerous**, but can  
be useful. My  
research code  
contains them

Static variables  
inside functions have  
memory  
They last between calls

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

int fun(int posnum)
{
    if (posnum <= 0)
    {
        exit(1); ←
    }
    // code that requires posnum > 0
    // ....
    // return some result
    int result = 10;
    return result;
}

int main(void)
{
    fun(-10);
    return 0;
}
```

whovian@phy504:~/host\$ gcc functions10.c -o functions10  
whovian@phy504:~/host\$ echo "exit code \$?"  
exit code 0  
whovian@phy504:~/host\$ ./functions10 ←  
whovian@phy504:~/host\$ echo "exit code \$?"  
exit code 1 ←

# Error Handling

Code should always check for errors constantly.

There are many ways to do it

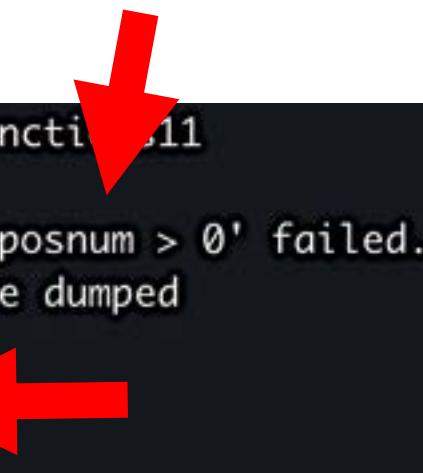
One is the “**let it crash**” approach. Code that crashes with exit code != 0 is usually much better than code that runs with wrong/undefined behavior

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

int fun(int posnum)
{
    assert (posnum > 0);

    // code that requires posnum > 0
    // ...
    // return some result
    int result = 10;
    return result;
}

int main(void)
{
    fun(-10);
    printf("End of main\n");
    return 0;
}
```



# Error Handling

C provides the assert function

Nice feature about assert: it provides the file name and line where the code failed

Assert also writes the check that failed. In this case **posnum > 0**

# Global Variables

Global variables are evil (sometimes unavoidable evil!)

```
#include <stdio.h>

int count; ← Everyone has
            access to
            Global
            Variables

int fun(void)
{
    count++;
    return count;
}

int main(void)
{
    int count = 0; █ I SEE YOU'RE USING GLOBAL
    printf("count = %d \n", fun());
    return 0;
}
```



# Global Variables

Global variables are evil (sometimes unavoidable evil!)

```
#include <stdio.h>

// source:
// https://www.geeksforgeeks.org/global-variables-in-c/

int a, b; // global variables

void add()
{
    printf("%d", a + b);
}

int main()
{
    a = 10;
    b = 15;
    add();
    return 0;
}
```

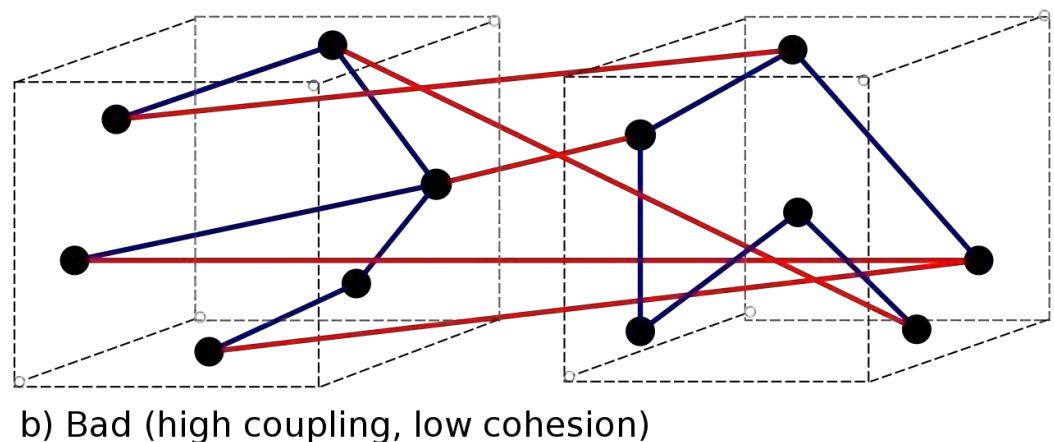
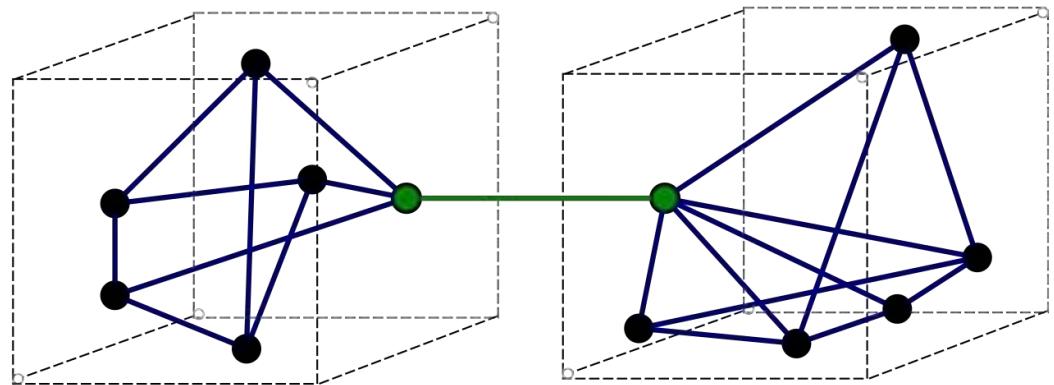
# Global Variables

## Why Global variables are Evil? Answers from SO

*“The problem with global variables is that since every function has access to these, it becomes increasingly hard to figure out which functions actually read and write these variables”* Source: <https://stackoverflow.com/a/485020/2472169>

*The problem that global variables create for the programmer is that it expands the inter-component coupling surface between the various components that are using the global variables.*

Source: <https://stackoverflow.com/a/24501533/2472169>



```
#include <stdio.h>

int main()
{
    enum day {
        sunday,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday,
        saturday
    };
    enum day d = thursday;
    printf("The day number stored in d is %d\n", d);
    enum day f = wednesday;
    printf("The day number stored in f is %d\n", f);

    if (d == f)
    {
        printf("d and f represent the same day\n");
    }
    else
    {
        printf("d and f represent different days\n");
    }
    return 0;
}
```

If you don't specify  
the vars values,  
compiler  
automatically write 0,  
1, 2, 3...

## Working with Structures:

### enum

enum is a set of integer variables with names that, in some ways, vaguely look like a type

C enum only holds integers

```
#include <stdio.h>
int main()
{
    enum day {
        sunday,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday,
        saturday
    };
    enum day d = thursday;
    printf("The day number stored in d is %d\n", d);
    enum day f = wednesday;
    printf("The day number stored in f is %d\n", f);

    if (d == f)
    {
        printf("d and f represent the same day\n");
    }
    else
    {
        printf("d and f represent different days\n");
    }
    return 0;
}
```

whovian@phy504:~/host/C\$ gcc enum1.c -o enum1  
whovian@phy504:~/host/C\$ ./enum1  
The day number stored in d is 4  
The day number stored in f is 3  
d and f represent different days

## Working with Structures:

### enum

C **enum** only  
holds **integers**

# Working with Structures: **enum**

```
#include <stdio.h>

// an interesting example from ChatGPT

enum Error {
    SUCCESS,
    FILE_NOT_FOUND,
    PERMISSION_DENIED,
    NETWORK_ERROR,
    OUT_OF_MEMORY
};

const char* error_messages[] = {
    "Success",
    "File not found",
    "Permission denied",
    "Network error",
    "Out of memory"
};

int main() {
    enum Error error = FILE_NOT_FOUND;

    printf("Error: %s\n", error_messages[error]);

    return 0;
}
```

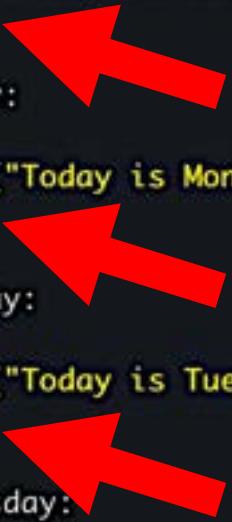
C **enum** can provide “names” to positions in arrays. An interesting application is for printing error messages

# Switch statements (with enums)

```
#include <stdio.h>
int main()
{
    enum day {
        sunday,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday,
        saturday
    };

    enum day e = wednesday; whovian@phy504:~/host/C$ gcc enum2.c -o enum2
    switch (e)           whovian@phy504:~/host/C$ ./enum2
    {                   Today is Wednesday
        case sunday:
        {
            printf("Today is Sunday\n");
            break;
        }
        case monday:
        {
            printf("Today is Monday\n");
            break;
        }
        case tuesday:
        {
            printf("Today is Tuesday\n");
            break;
        }
        case wednesday:
    }
```

**{ } are not mandatory, but I always include them (they isolate local variables you may create inside each case)**



# enums are a bit weird

C enums are not strongly typed ([a mistake that C++ fixed](#))

OK, first example: old-style enums do not have their own scope:

```
enum Animals {Bear, Cat, Chicken};      This is C
enum Birds {Eagle, Duck, Chicken}; // error! Chicken has already been declared!

enum class Fruits { Apple, Pear, Orange };
enum class Colours { Blue, White, Orange }; // no problem!
```

Second, they implicitly convert to integral types, which can lead to strange behaviour:

```
bool b = Bear && Duck; // what?          This is bad!!!
```

Because of these problems I avoid C enums,  
but I am perfectly happy with C++ enums

# typedef keyword

Assign an alternate name to a data type

```
#include <stdio.h>

typedef int chicken;
int main (void)
{
    chicken x = 100;
    printf ("Physics 504 is fun. Your grade %i\n", x);
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc typedef.c -o typedef
whovian@phy504:~/host/C$ ./typedef
Physics 504 is fun. Your grade 100
```

```

#include <stdio.h>
#include <math.h>

typedef struct
{
    double x;
    double y;
} Point;

double distance (Point p1, Point p2)
{
    const double diffx = p1.x - p2.x;
    const double diffy = p1.y - p2.y;
    return sqrt (diffx * diffx + diffy * diffy);
}

int main ()
{
    Point p1;
    p1.x = 0.0;
    p1.y = 1.0;

    Point p2;
    p2.x = 1.0;
    p2.y = 0.0;

    printf("P1 = (%.2f, %.2f)\n", p1.x, p1.y);
    printf("P2 = (%.2f, %.2f)\n", p2.x, p2.y);
    printf("|P1-P2| = %.4f \n", distance(p1, p2));
}

```

# structs in C

structs allows C to do  
Object Oriented  
Program (OOP)

C++ will take structs  
to another level with  
Classes

```

whovian@phy504:~/host$ gcc struct.c -o struct -lm
whovian@phy504:~/host$ ./struct
P1 = (0.00, 1.00)
P2 = (1.00, 0.00)
|P1-P2| = 1.4142

```

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

typedef struct
{
    char name[100];
    int day;
    int month;
    int year;
} Date;

void print_date(Date x)
{
    printf("Name: %s, Birthday: %.2d-%2d-%4d\n",
           x.name, x.month, x.day, x.year);
}

int main ()
{
    Date birthdays[2];
    strcpy(birthdays[0].name, "Vivian Miranda");
    birthdays[0].day = 28;
    birthdays[0].month = 02;
    birthdays[0].year = 1986;

    strcpy(birthdays[1].name, "Taylor Swift");
    birthdays[1].day = 13;
    birthdays[1].month = 12;
    birthdays[1].year = 1989;

    print_date(birthdays[0]);
    print_date(birthdays[1]);
}
```

# structs in C

structs works like any type. For example, you can make arrays

PS: why not

**birthday[0].name = “Viv”?**

array type is not assignable

```
whovian@phy504:~/host$ gcc struct2.c -o struct2
whovian@phy504:~/host$ ./struct2
Name: Vivian Miranda, Birthday: 02-28-1986
Name: Taylor Swift, Birthday: 12-13-1989
```

# structs in C

Be careful with overflow – undefined behavior

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

typedef struct
{
    char name[10];
    int day;
    int month;
    int year;
} Date;

void print_date(Date x)
{
    printf("Name: %s, Birthday: %.2d-%2d-%4d\n",
           x.name, x.month, x.day, x.year);
}

int main ()
{
    Date birthdays[2];
    strcpy(birthdays[0].name, "Vivian Miranda Bla bla bla bla bla bla ok");
    birthdays[0].day = 28;
    birthdays[0].month = 02;
    birthdays[0].year = 1986;

    print_date(birthdays[0]);
}
```

```
whovian@phy504:~/host$ gcc struct2.c -o struct2
whovian@phy504:~/host$ ./struct2
Name: Vivian Miran, Birthday: 02-28-1986
```



```
#include <stdio.h>

typedef struct
{
    char name[100];
    int day;
    int month;
    int year;
} Date;

void print_date(Date x)
{
    printf("Name: %s, Birthday: %.2d-%2d-%4d\n",
           x.name, x.month, x.day, x.year);
}

int main ()
{
    Date test = {"Vivian Miranda", 28, 02, 1986};
    print_date(test);

/* Code below does not work, it is not initialization
   Code below is assignment.
   Initialization != assignment
   Date birthdays[2];
   birthdays[0] = {"Vivian Miranda", 28, 02, 1986};
*/
}
```

# structs in C

Initialization  
not the same  
as assignment

# structs in C OOP style

```
#include <stdio.h>

typedef struct
{
    char name[100];
    int day;
    int month;
    int year;
} Date;

int get_day(const Date* const self)
{
    return self->day;
}

void set_day(Date* const self, const int day)
{
    self->day = day;
}

int main()
{
    Date x;
    set_day(&x, 28);
    printf("Date = %.2d\n", get_day(&x));
}
```

In OOP, we don't change member variables directly. We use functions

First time seeing pointers and the ability to change function arguments passed by value

# Nested structs

```
typedef struct
{
    int day;
    int month;
    int year;
} Date;

typedef struct
{
    char name[100];
    Date bday;
} Student;

int main ()
{
    Student x;
    strcpy(x.name, "Vivian Miranda");
    x.bday.day = 28;
    x.bday.month = 2;
    x.bday.year = 1986;
    printf("Date = %.2d\n", x.bday.day);
```

Nested struct is a good way to make struct manageable

```
whovian@phy504:~/host$ gcc struct5.c -o struct5
whovian@phy504:~/host$ ./struct5
Date = 28
```

# Unions and anonymous Unions

```
#include <stdio.h>

typedef struct
{
    union
    {
        struct
        {
            double x;
            double y;
            double z;
        };
        double raw[3];
    };
} vec3d;

int main ()
{
    vec3d v;
    v.x = 4.0;
    v.raw[1] = 3.0; // Equivalent to v.y = 3.0
    v.z = 2.0;

    printf("(x,y,z) = (%.2f,%.2f,%.2f) \n", v.x, v.y, v.z);
    // an alternative way to get the same data
    printf("(x,y,z) = (%.2f,%.2f,%.2f) \n", v.raw[0], v.raw[1], v.raw[2]);
}
```

advanced  
feature:  
types stored  
in the same  
memory  
location

```
whovian@phy504:~/host$ gcc union1.c -o union1
whovian@phy504:~/host$ ./union1
(x,y,z) = (4.00,3.00,2.00)
(x,y,z) = (4.00,3.00,2.00)
```

```
#include <stdio.h>

typedef struct
{
    int rows;
    int cols;
    double data[10000];
} Matrix;

// implement the following functions

// double get_elems(Matrix a, const int row, const int col)
// that returns a(i,j).
// The function should check if row and col are out of bounds

// void set_elems(Matrix a, const int row, const int col)
// The function should check if row and col are out of bounds

// Matrix msum(Matrix a, Matrix b)
// The function should check if a and b have same rows and cols

// Matrix mmult(Matrix a, Matrix b)
// The function should check if a can be multiplied by b

int main()
{
    return 0.0;
}
```

# Homework

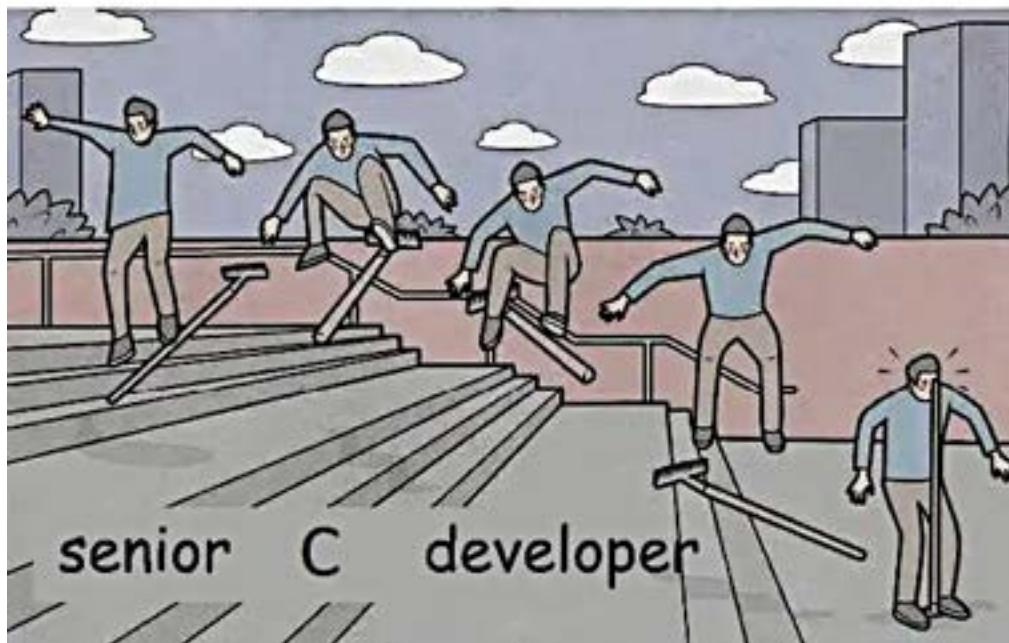
# Lecture 14: C part V



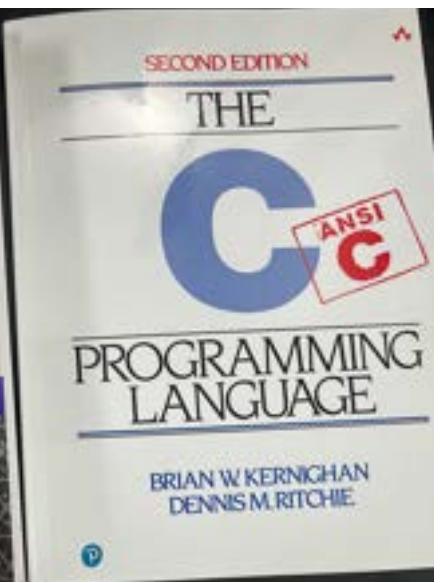
junior C developer

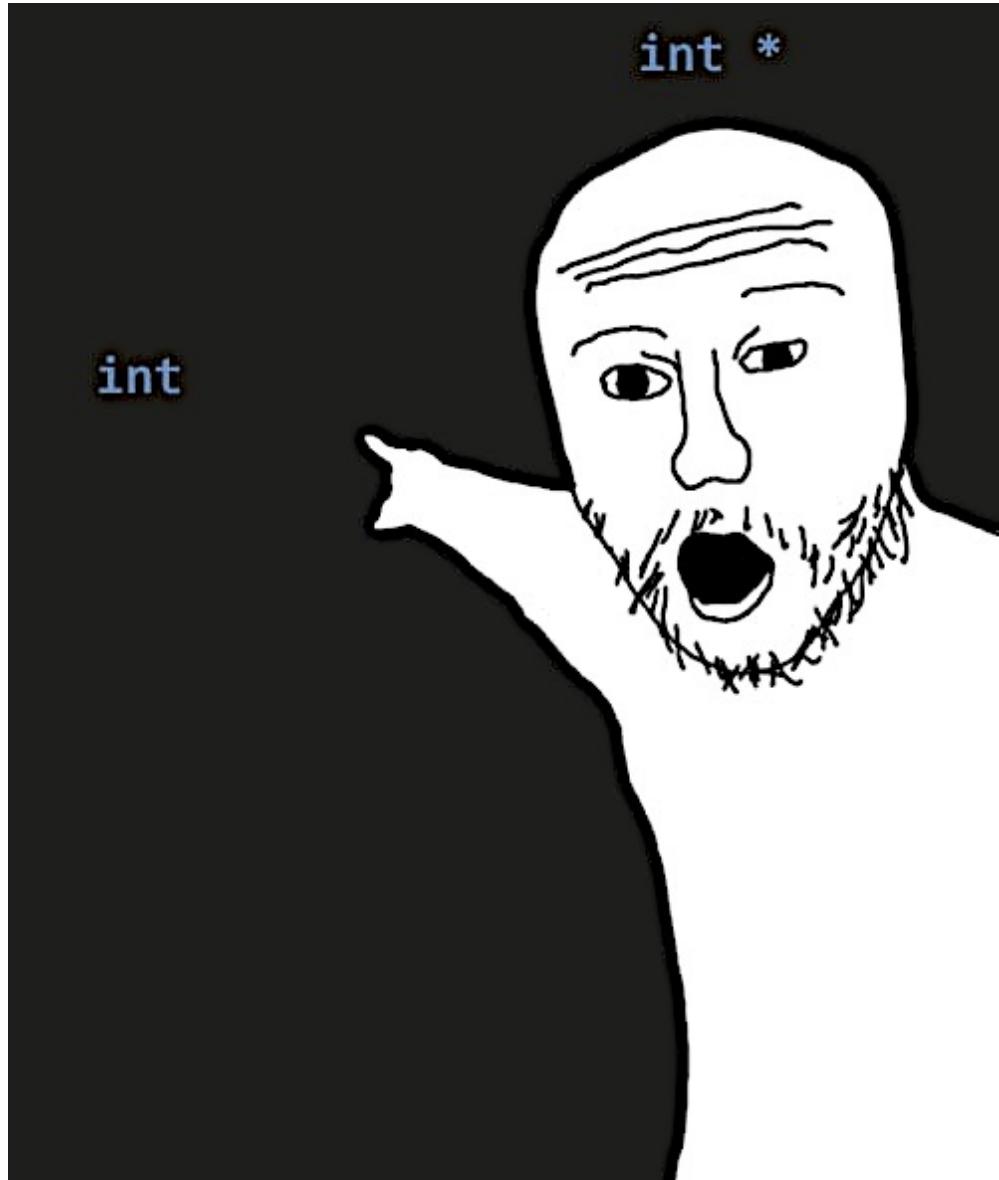


## Suggested Literature



senior C developer





pointers in C

# Pointers

Pointers point to the memory location of a variable



```
#include <stdio.h>
int main ()
{
    int x = 10;
    int* y = &x;

    printf("x = %d\n", x);
    printf("The value of pointer y = %d\n", *y);
    printf("The address of x = %p\n", (void*) &x);
    printf("The address of y = %p\n", (void*) y);
}
```

whovian@phy504:~/host/C\$ gcc pointers1.c -o pointers  
whovian@phy504:~/host/C\$ ./pointers  
x = 10  
The value of pointer y = 10  
The address of x = 0x400180539c  
The address of y = 0x400180539c

indirection operator 

# Memory mapping

Mapping is written in Hexadecimals

0xNum implies Num is in Hexadecimal

```
whovian@phy504:~/host/C$ gcc pointers1.c -o pointers
whovian@phy504:~/host/C$ ./pointers
x = 10
The value of pointer y = 10
The address of x = 0x400180539c
The address of y = 0x400180539c
```

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

convenience: nums shorter and easy to convert to binary

Conversion from [0-F] to binary takes 4 bits

Therefore, 0xE2 =  $14 * 16 + 2 = \underline{\text{1110}} \underline{\text{0010}}$  (binary)

# Memory mapping

Mapping is written in Hexadecimals

0xNum implies Num is in Hexadecimal

```
whovian@phy504:~/host/C$ gcc pointers1.c -o pointers
whovian@phy504:~/host/C$ ./pointers
x = 10
The value of pointer y = 10
The address of x = 0x400180539c
The address of y = 0x400180539c
```

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

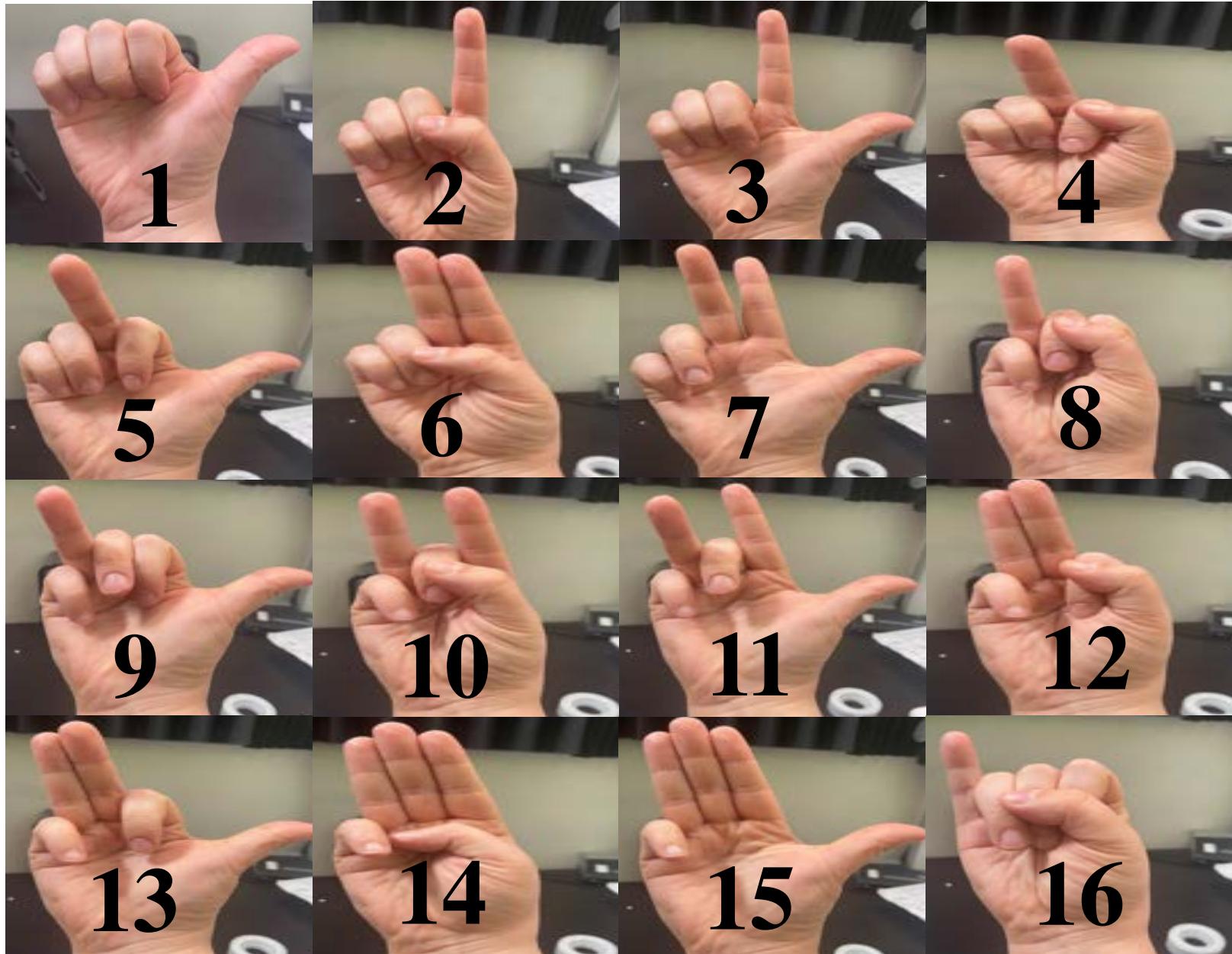
More complicated: 0xAE0 to binary

Conversion from [0-F] to binary takes 4 bits

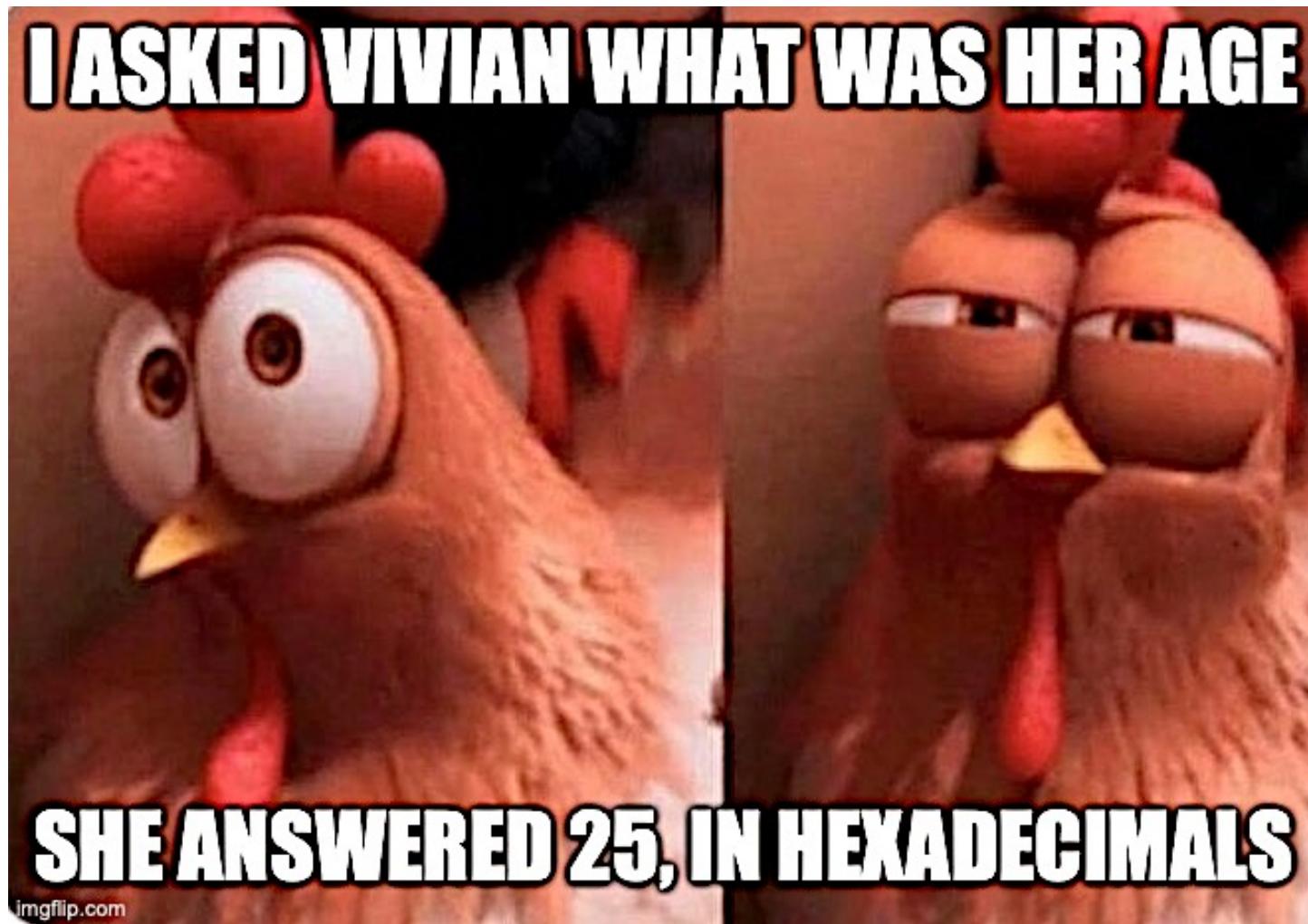
Therefore, 0xAE2 = 1010 1110 0010 (binary)

A      E      2

# Practice 4 bits binary with one hand (up to 15!)



I only answer my age in Hexadecimals



# Pointers

Pointers keep track of the value of the variable it points to

```
#include <stdio.h>

int main ()
{
    int x = 10;
    int* y = &x;
    int z = *y;

    x++;

    printf("x = %d\n", x);
    printf("y = %d\n", *y);
    printf("z = %d\n", z);
}
```

```
whovian@phy504:~/host/C$ ./pointers2
x = 11
y = 11
z = 10
```

How?

**int\* y = &x;**

copies de address in  
memory not the  
value

# Pointers

Pointers keep track of the value of the variable it points to

```
#include <stdio.h>
int main ()
{
    int x = 10;
    int* y = &x;
    int z = *y;
    x++;

    printf("x = %p\n", (void*) &x);
    printf("y = %p\n", (void*) y);
    printf("z = %p\n", (void*) &z);
}
```

whovian@phy504:~/host/C\$ gcc pointers3.c -o pointers3  
whovian@phy504:~/host/C\$ ./pointers3  
x = 0x4001805398 ←  
y = 0x4001805398 ←  
z = 0x400180539c

How?  
**int\* y = &x;**  
copies de  
address in  
memory not the  
value

# Pointers

```
#include <stdio.h>
int main (void)
{
    char c = 'Q';
    char* ptr = &c;

    printf ("%c %c\n", c, *ptr);

    c = '/';
    printf ("%c %c\n", c, *ptr);

    *ptr = '(';
    printf ("%c %c\n", c, *ptr);

    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc pointers4.c -o pointers4
whovian@phy504:~/host/C$ ./pointers4
```

Q Q  
//  
()



Pointers can change  
the value of the  
original value as well

They are “in sync”  
because they get the  
value stored in the  
same memory  
location

```
#include <stdio.h>

typedef struct
{
    char name[100]
    int day;
    int month;
    int year;
} Date;

void print_date(Date x)
{
    printf("Name: %s, Birthday: %.2d-%2d-%4d\n",
           x.name, x.month, x.day, x.year);
}

int main ()
{
    Date test = {"Vivian Miranda", 28, 02, 1986};
    print_date(test);

    Date* ptr = &test;
    print_date(*ptr);
}
```

# Pointers

The same logic applies to **structs**

Pointer point to the memory location of the **struct**

This is important for optimization.

Copying **structs** can be expensive

# Pointers

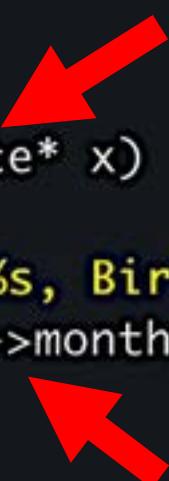
```
#include <stdio.h>

typedef struct
{
    char name[100];
    int day;
    int month;
    int year;
} Date;

void print_date(Date* x)
{
    printf("Name: %s, Birthday: %.2d-%2d-%4d\n",
           x->name, x->month, x->day, x->year);
}

int main ()
{
    Date test = {"Vivian Miranda", 28, 02, 1986};
    print_date(&test);
}
```

Copy the address by value (not the contents of the struct)



Use the symbol ->

This is important for optimization

Copying structs can be expensive

```
#include <stdio.h>

typedef struct
{
    char name[100];
    int day;
    int month;
    int year;
} Date;

void test1(Date x)
{
    x.year = 2000;
}

void test2(Date* x)
{
    x->year = 2000;
}

void print_date(Date* x)
{
    printf("Name: %s, Birthday: %.2d-%2d-%4d\n",
           x->name, x->month, x->day, x->year);
}

int main()
{
    Date test = {"Vivian Miranda", 28, 02, 1986};
    test1(test);
    print_date(&test);
    test2(&test);
    print_date(&test);
}
```

Copied the  
address

# Pointers

Passing the address  
by value allow  
functions to modify  
the content of its  
arguments

```

#include <stdio.h>

void test(int* const ptr)
{
    *ptr = 1;
    // Assigning ptr to point to another
    // address is not allowed. Example
    // ptr = NULL; // not allowed!!!
}

int main ()
{
    int x = 4;
    int* const y = &x;
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf(" y = %p\n", (void*) y);

    test(y);
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf(" y = %p\n", (void*) y);
}

```

# Pointers

Passing the address by value allow functions to modify the content of its arguments

This can be good or bad

How do we prevent this?

**const correctness**

```

whovian@phy504:~/host/C$ gcc pointers8.c -o pointers8
whovian@phy504:~/host/C$ ./pointers8
x = 4
&x = 0x400180539c ← Address
y = 0x400180539c
x = 1
&x = 0x400180539c ← unchanged
y = 0x400180539c

```

Address  
unchanged

```

#include <stdio.h>

const int z = 10;

void test(const int* ptr)
{
    ptr = &z;
    // Assigning (*ptr) to another
    // value is not allowed. Example
    // *ptr = 1; // not allowed!!!

    // However since ptr was passed
    // by value, the change will not
    // propagate outside the function
    printf(" ptr = %p\n", (void*) ptr);
}

int main()
{
    int x = 4;
    const int* y = &x;
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf(" y = %p\n", (void*) y);

    test(y);
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf(" y = %p\n", (void*) y);
    printf("&z = %p\n", (void*) &z);
}

```

# Pointers

```

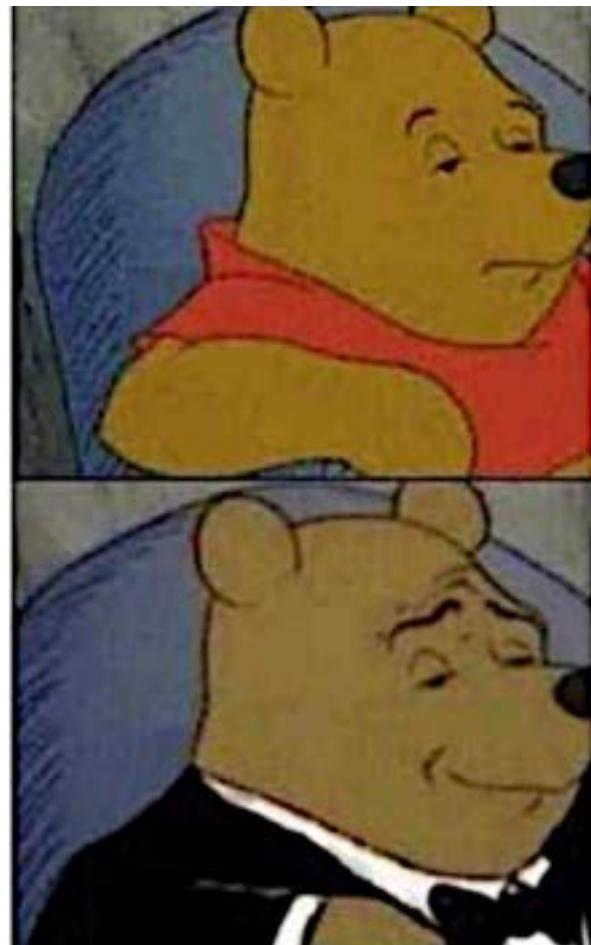
whovian@phy504:~/host/C$ gcc pointers9.c -o pointers9
whovian@phy504:~/host/C$ ./pointers9
x = 4
&x = 0x400180539c
y = 0x400180539c
ptr = 0x4000002004 ←
x = 4
&x = 0x400180539c
y = 0x400180539c ←
&z = 0x4000002004

```

Passing the address by value  
allow functions to modify the  
content of its arguments (but  
not the pointer itself)

What if we need to modify the  
pointer itself? We need pointer  
of pointers!

# Dynamically allocated memory



**Int a[10];**

**Int\* a = (int\*)malloc(10\*sizeof(int));**

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

int main ()
{
    const int sz = 10;
    double* x = (double*) malloc(sz*sizeof(double));
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for(int i=0; i<sz; i++)
    {
        x[i] = i;
    }
    for(int i=0; i<sz; i++)
    {
        printf("x[%d] = %0.2f\n", i, x[i]);
    }
    // dynamically allocated memory is persistent
    // it only deallocates when either the program
    // ends or when you free the pointers
    free(x);

    return 0;
}
```

always check if  
the pointer is  
NULL

Dynamically  
allocated  
memory

malloc provides  
memory but does  
not initialize  
array values  
can't forget to  
free the memory

```
whovian@phy504:~/host/Phy504Docker/C$ ./pointers12
x[0] = 0.00
x[1] = 1.00
x[2] = 2.00
x[3] = 3.00
x[4] = 4.00
x[5] = 5.00
x[6] = 6.00
x[7] = 7.00
x[8] = 8.00
x[9] = 9.00
```

# Dynamically allocated memory

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

int main ()
{
    const int sz = 10;
    double* x = (double*) calloc(sz, sizeof(double));
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for(int i=0; i<sz; i++)
    {
        // ok because calloc init all vars to zero
        printf("x[%d] = %0.2f\n", i, x[i]);
    }
    // dynamically allocated memory is persistent
    // it only deallocates when either the program
    // ends or when you free the pointers
    free(x);

    return 0;
}
```

Both malloc and calloc  
return void\*. So  
casting is needed

malloc provides  
memory but does  
not initialize array  
values

calloc does init  
the array to zero

malloc and calloc  
have different  
arguments (1 x 2)

Can't forget to  
free the memory

# Memory leak with malloc

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

int main ()
{
    const int sz = 200000000;
    // DANGEROUS CODE - THIS WILL CRASH YOUR PC!
    for(int i=0; i<50; i++)
    {
        double* x = (double*) malloc(sz*sizeof(double));
        if (x == NULL)
        {
            printf("Error in memory: memory exhaustion \n");
            exit(1);
        }
        x[0] = 10;
        sleep(1);
        x = NULL; // reference to the memory lost but
                   // the memory continues to be allocated
    }
    return 0;
}
```

**Memory leak**



# Additional Reading

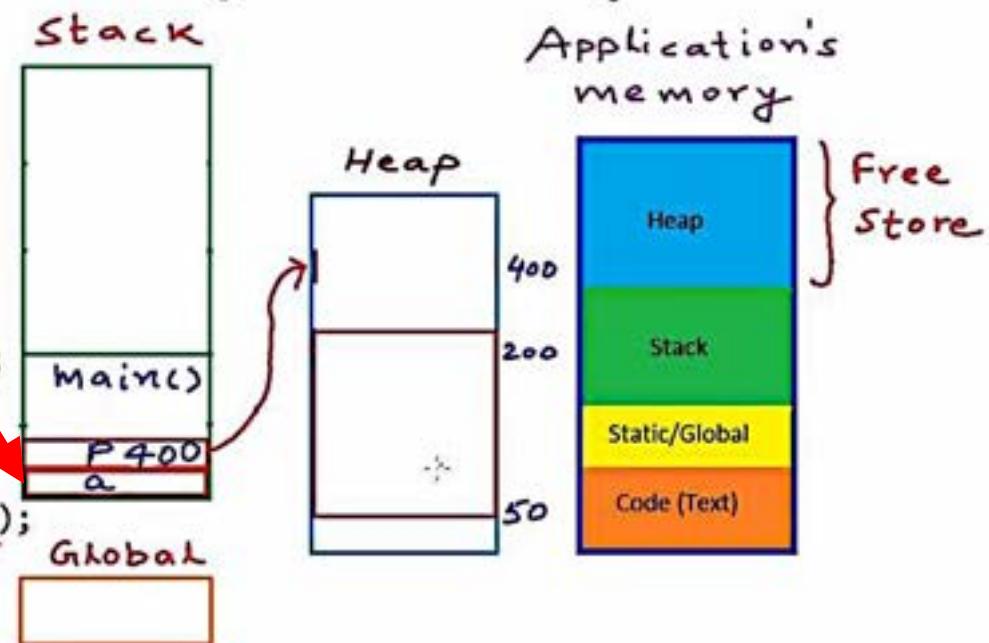
- Top 20 pointer mistakes: <https://www.acodersjourney.com/top-20-c-pointer-mistakes/>

# Additional Reading: why do we need malloc?

Malloc allow programs to access a large pool of memory: **heap**

Without **malloc**, we only have access to stack automatic memory (fast but small)

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a; // goes on stack
    int *p;
    p = (int*)malloc(sizeof(int));
    *p = 10;
    free(p);
    p = (int*)malloc(20*sizeof(int));
}
```



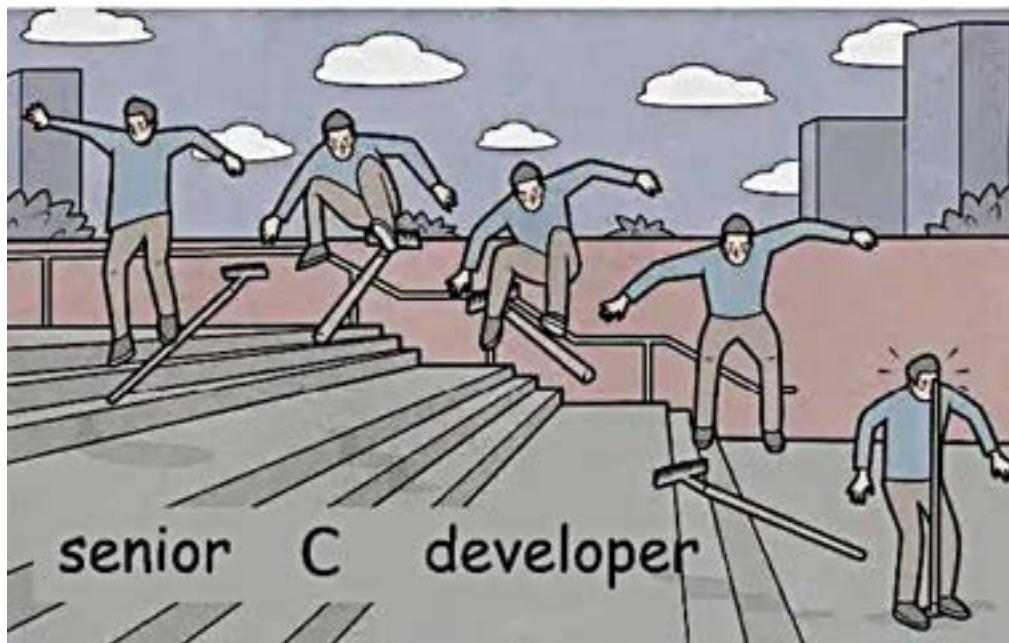
# Lecture 15: C part VI



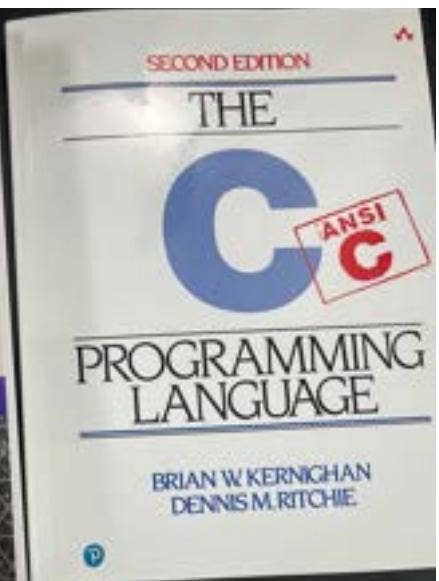
junior C developer



## Suggested Literature



senior C developer





Double pointers in  
C

# Pointers

```
#include <stdio.h>

int z = 10;

void test(int** ptr)
{
    *ptr = &z;
}

int main()
{
    int x = 4;      // value
    int* y = &x;   // pointer to int
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf("y = %p\n", (void*) y);
    printf("&z = %p\n", (void*) &z);

    test(&y);
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf("y = %p\n", (void*) y);
    printf("&z = %p\n", (void*) &z);
}
```

The address of  
the pointer to int

is passed by

```
whovian@phy504:~/host/Phy504Docker/C$ gcc poi
whovian@phy504:~/host/Phy504Docker/C$ ./point
x = 4
&x = 0x400180537c
y = 0x400180537c
x = 4
&x = 0x400180537c
y = 0x4000004010
&z = 0x4000004010
```

What if we need to  
modify the pointer  
function argument?

We need pointer of  
pointers!

# Pointers

```
#include <stdio.h>

int main()
{
    int a = 5;
    int* ptr = &a;
    int** d_ptr = &ptr;

    printf("Size of normal Pointer: %li \n",
           sizeof(ptr));

    printf("Size of Double Pointer: %li \n",
           sizeof(d_ptr));

    return 0;
}
```

**int\*** and **int\*\***  
have the same size,  
they are just both  
addresses

```
whovian@phy504:~/host/C$ gcc pointers18.c -o pointers18
whovian@phy504:~/host/C$ ./pointers18
Size of normal Pointer: 8
Size of Double Pointer: 8
```

**8 bytes = 64 bits =  
up to 16 digits in  
Hex**

# Constant correctness on double pointers is involved

- The double pointer to an `int` is constant, i.e., it can't be assigned to another pointer to int address: `int** const ptr;`
- The pointer to `int` that the double pointer holds (`*ptr`) is not a constant pointer to an integer. (`*ptr`) can be assigned to another integer address
- What if we don't want the pointer (`*ptr`) to `int` that the double pointer holds to be assigned to another address)? `int* const * const`
- Even in this case, the `int` that `*(*ptr)` holds can be modified? What if we don't want that? We must define the double pointer `ptr` as `const int* const * const` or `int const * const * const` (some people find that easier to remember. In this case, `const` is a promise for the operator on the left

```
#include <stdio.h>

const int z = 10;

void test(const int* const * const ptr)
{
    *ptr = &z; // This fails!
    ptr = NULL; // This also fails!
    (*ptr) = 4; // This also fails!
}

int main ()
{
    int x = 4;
    const int* const y = &x;
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf(" y = %p\n", (void*) y);

    test(&y);
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf(" y = %p\n", (void*) y);
    printf("&z = %p\n", (void*) &z);
}
```

Pointers  
The double pointer  
to an **int**, **ptr**, is  
constant  
  
The pointer **ptr** can't  
be assigned to  
another pointer to int  
address

```
pointers11.c: In function 'test':
pointers11.c:7:10: error: assignment of read-only location '*ptr'
    7 |     *ptr = &z; // This fails!
      |     ^
pointers11.c:8:9: error: assignment of read-only parameter 'ptr'
    8 |     ptr = NULL; // This also fails!
      |     ^
pointers11.c:9:13: error: assignment of read-only location '**ptr'
    9 |     (*ptr) = 4; // This also fails!
      |     ^
```

```

#include <stdio.h>

const int z = 10;

void test(const int* const * const ptr)
{
    *ptr = &z; // This fails!
    ptr = NULL; // This also fails!
    *(*ptr) = 4; // This also fails!
}

int main()
{
    int x = 4;
    const int* const y = &x;
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf("y = %p\n", (void*) y);

    test(&y);
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf("y = %p\n", (void*) y);
    printf("&z = %p\n", (void*) &z);
}

```

Pointers  
The pointer to **int** that  
the double pointer  
holds, **(\*ptr)**, is  
**constant**

The pointer **(\*ptr)**  
can't be assigned to  
another integer  
address

```

pointers11.c: In function 'test':
pointers11.c:7:10: error: assignment of read-only location '*ptr'
  7 |     *ptr = &z; // This fails!
   |
   |
pointers11.c:8:9: error: assignment of read-only parameter 'ptr'
  8 |     ptr = NULL; // This also fails!
   |
   |
pointers11.c:9:13: error: assignment of read-only location '**ptr'
  9 |     *(*ptr) = 4; // This also fails!
   |

```

```
#include <stdio.h>

const int z = 10;

void test(const int* const * const ptr)
{
    *ptr = &z; // This fails!
    ptr = NULL; // This also fails!
    *(*ptr) = 4; // This also fails!
}

int main ()
{
    int x = 4;
    const int* const y = &x;
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf(" y = %p\n", (void*) y);

    test(&y);
    printf("x = %d\n", x);
    printf("&x = %p\n", (void*) &x);
    printf(" y = %p\n", (void*) y);
    printf("&z = %p\n", (void*) &z);
}
```

Pointers  
The integer that  
**\*(\*ptr)** holds can't  
be modified

The integer variable  
**\*(\*ptr)** can't be  
assigned to another  
integer value

```
pointers11.c: In function 'test':
pointers11.c:7:10: error: assignment of read-only location '*ptr'
    7 |     *ptr = &z; // This fails!
        |
pointers11.c:8:9: error: assignment of read-only parameter 'ptr'
    8 |     ptr = NULL; // This also fails!
        |
pointers11.c:9:13: error: assignment of read-only location '**ptr'
    9 |     *(*ptr) = 4; // This also fails!
        |
```

# Pointers

The trick to reading declarations is to read from right to left. Thus:

`int const`

A constant integer

Yes! **const int x;** is equivalent to **int const x;**

`int const *`

A (variable) pointer to a constant integer

`int * const`

A constant pointer to a (variable) integer

`int * const *`

A pointer to a constant pointer to an integer

`int const * *`

A pointer to a pointer to a constant integer

`int const * const *`

A pointer to a constant pointer to a constant integer

You can see that the `const` always refers to the text to its left, just as the `*` does.

Source: 21st Century C,  
2nd Edition. Author:  
Ben Klemens

# What about triple pointers in C?



# Dynamically allocated memory



**Int a[10];**

```
Int* a = (int*)malloc(10*sizeof(int));
```

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

int main ()
{
    const int sz = 10;
    double* x = (double*) malloc(sz*sizeof(double));
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for(int i=0; i<sz; i++)
    {
        x[i] = i;
    }
    for(int i=0; i<sz; i++)
    {
        printf("x[%d] = %0.2f\n", i, x[i]);
    }
    // dynamically allocated memory is persistent
    // it only deallocates when either the program
    // ends or when you free the pointers
    free(x);

    return 0;
}
```

always check if  
the pointer is  
NULL

Dynamically  
allocated  
memory

**malloc** provides  
memory but does  
not initialize array  
values

Can't forget to  
**free** the memory

```
whovian@phy504:~/host/Phy504Docker/CS ./pointers12
x[0] = 0.00
x[1] = 1.00
x[2] = 2.00
x[3] = 3.00
x[4] = 4.00
x[5] = 5.00
x[6] = 6.00
x[7] = 7.00
x[8] = 8.00
x[9] = 9.00
```

# Dynamically allocated memory

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

int main ()
{
    const int sz = 10;
    double* x = (double*) malloc(sz*sizeof(double));
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for(int i=0; i<sz; i++)
    {
        x[i] = i;
    }
    for(int i=0; i<sz; i++)
    {
        printf("x[%d] = %0.2f\n", i, x[i]);
    }
    // dynamically allocated memory is persistent
    // it only deallocates when either the program
    // ends or when you free the pointers
    free(x);

    return 0;
}
```

Why do we need **sizeof(double)**? Well, **void\*** is usually 2 bytes in size (1 byte = 8 bits)

You don't want to create an array of 2 bytes elements. You want an array of 8 bytes (**double**) elements

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

int main ()
{
    const int nrows = 10;
    const int ncols = 10;
    double** x = (double**) malloc(sizeof(double*)*nrows);
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for (int i=0; i<nrows; i++)
    {
        x[i] = (double*) calloc(ncols, sizeof(double));
        if (x[i] == NULL)
        {
            printf("Error in memory: memory exhaustion \n");
            exit(1);
        }
    }
    for (int i=0; i<nrows; i++)
    {
        for(int j=0; j<ncols; j++)
        {
            printf("x[%d,%d] = %0.2F", i, j, x[i][j]);
        }
        printf("\n");
    }
    for (int i=0; i<nrows; i++)
    {
        free(x[i]);
    }
    free(x);
    return 0;
}
```

# malloc & calloc on double pointers

In this common way:  
we need to **malloc** each dimension separately  
(with the **for loop**)

When deallocated  
memory with **free**, we  
also need to free each dimension separately

# **malloc & calloc** on double pointers

The problem with that approach is the memory is not contiguous

In normal circumstances, that is totally ok

But there are more optimal ways in High-Performance Computing (contiguous memory)

Solution 1: 1D array such that **a[i,j] = v[i + nrows\*j]**

Solution2: (ref: Ex4 - <https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/>)

# malloc & calloc on double pointers

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

int main ()
{
    const int nrows = 3;
    const int ncols = 2;
    double* x = (double*) malloc(sizeof(double)*ncols*nrows);
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for (int i=0; i<nrows; i++)
    {
        for(int j=0; j<ncols; j++)
        {
            x[i + nrows*j] = i + j;
            printf("x[%d,%d] = %0.2f  ", i, j, x[i + nrows*j]);
        }
        printf("\n");
    }
    free(x);
    return 0;
}
```

Solution 1: 1D array such that  
 $a[i,j] = v[i + nrows*j]$

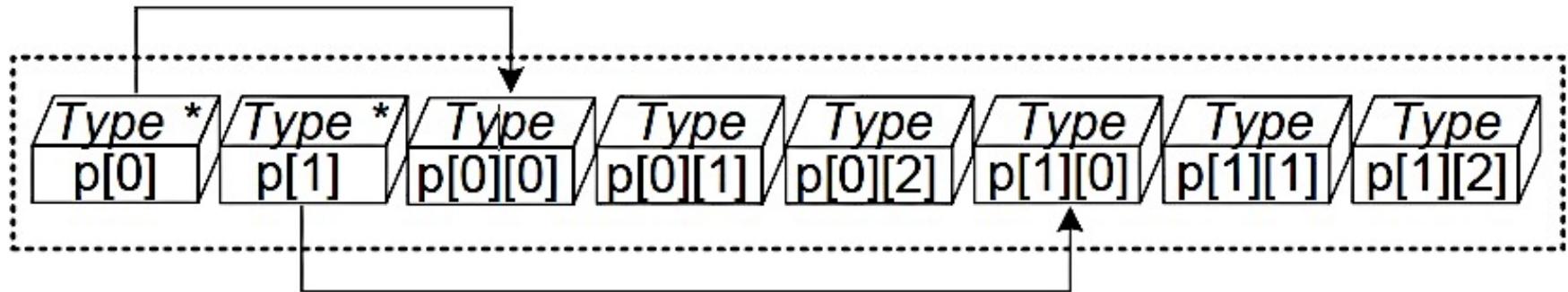
Cumbersome!

Good part:  
memory is  
contiguous

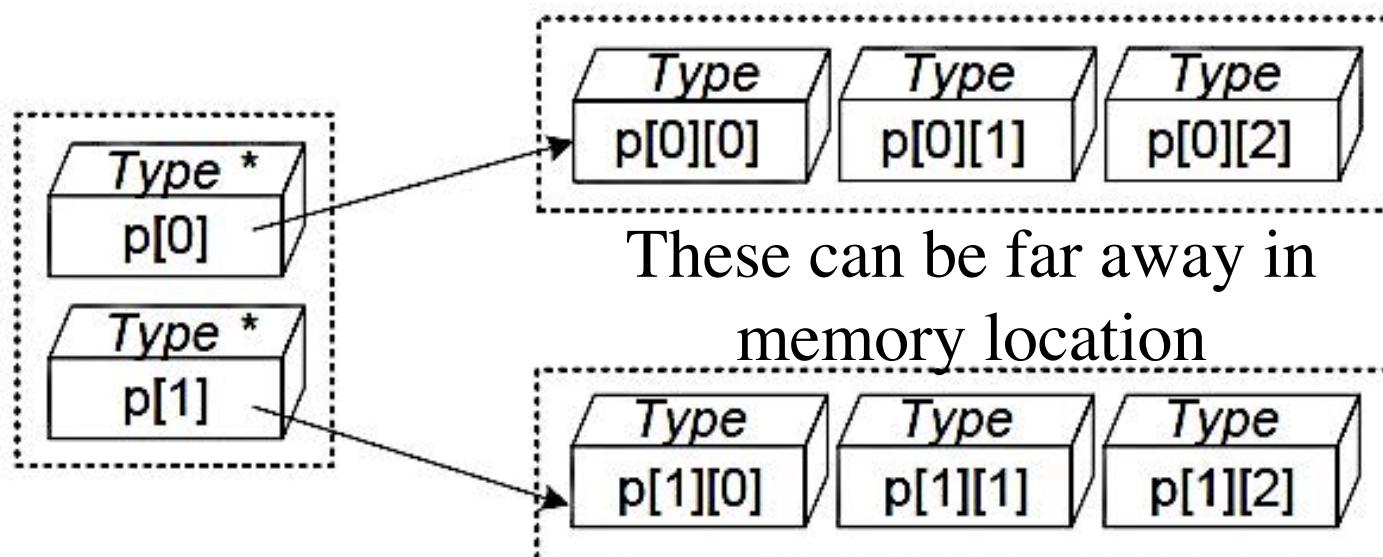
```
whovian@phy504:~/host/C$ gcc pointers17.c -o pointers17
whovian@phy504:~/host/C$ nano pointers17.c
whovian@phy504:~/host/C$ ./pointers17
x[0,0] = 0.00  x[0,1] = 1.00
x[1,0] = 1.00  x[1,1] = 2.00
x[2,0] = 2.00  x[2,1] = 3.00
```

# malloc & calloc on double pointers

good: Memory is contiguous



bad: Memory is non-contiguous. Integer arithmetic w/  
pointers not possible, i.e., **a[i,j] != \*(a + i + nrows\*j)**



# malloc & calloc on double pointers

pointer arithmetic

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

int main ()
{
    const int nrows = 3;
    const int ncols = 3;
    const int len = sizeof(double*)*nrows + sizeof(double)*ncols*nrows;
    double** x = (double**) malloc(len);
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for (int i=0; i<nrows; i++)
    { // for loop to point rows pointer to right loc in 2D array
        x[i] = ((double*)(x + nrows) + ncols*i);
    }

    for (int i=0; i<nrows; i++)
    {
        for(int j=0; j<ncols; j++)
        {
            x[i][j] = i + j;
            printf("x[%d,%d] = %0.2f ", i, j, x[i][j]);
        }
        printf("\n");
    }
    free(x);
    return 0;
}
```

Solution 2  
(awesome)  
Memory is  
contiguous

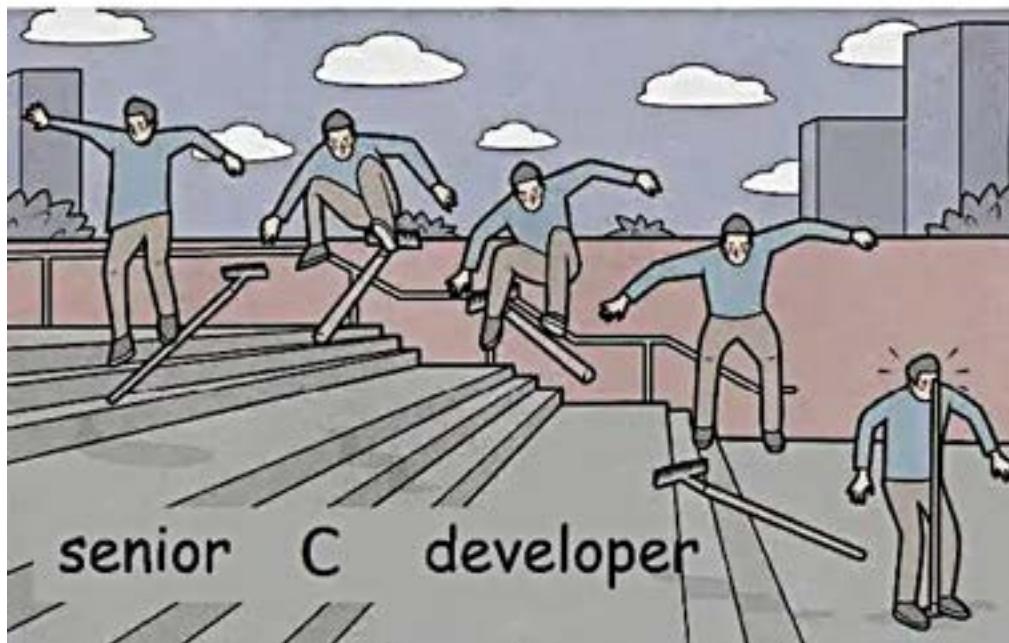
# Lecture 16: C part VII



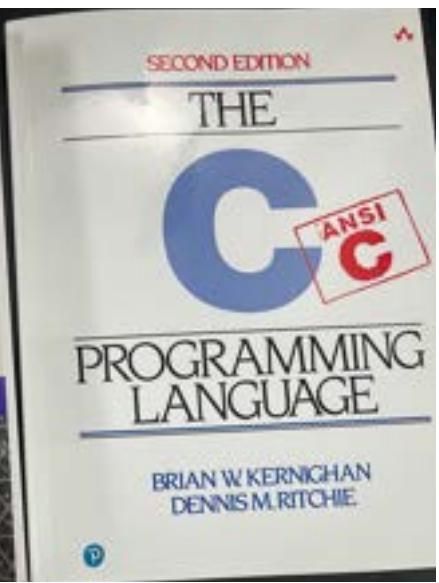
junior C developer



Suggested Literature



senior C developer



# Pointer Arithmetic

```
#include <stdio.h>

// pointers are incremented and decremented
// by the size of the data type they point to
int main()
{
    int a = 22;
    int* p = &a;
    printf("p    = %p\n", (void*) p);
    printf("p++ = %p\n\n", (void*) ++p); // +4bytes
                                         // = +16bits
                                         // = +0x4

    double b = 22.22;
    double* q = &b;
    printf("q    = %p\n", (void*) q);
    printf("q++ = %p\n\n", (void*) ++q); // +8 bytes
                                         // = +32bits
                                         // = +0x8

    char c = 'a';
    char *r = &c;
    printf("r    = %p\n", r);
    printf("r++ = %p\n", (void*) ++r); // +1 byte
                                         // = +4bits
                                         // = +0x1

    return 0;
}
```

Pointer arithmetic is super useful to speed-up access to elements stored on contiguous containers

```
whovian@phy504:~/hos
p    = 0x4001805384
p++ = 0x4001805388

q    = 0x4001805388
q++ = 0x4001805390

r    = 0x4001805383
r++ = 0x4001805384
```

Source:

<https://www.geeksforgeeks.org/pointer-arithmetics-in-c-with-examples/>

# Pointer Arithmetic

```
#include <stdio.h>

int main()
{
    int x[10] = {1,4,6,8,10,12,14,16,18,20};
    int* ptr1 = &x[0]; // address of the first elem

    printf("x[%i] = %i \n", 4, *(ptr1+3)); // 4th elem

    int *ptr2 = &x[9];
    printf("proof x mem is contiguous: 0x%lx\n",
           (ptr2 - ptr1)*sizeof(int)); // 9 * 4 = 36 (dec)
                                    // = 24 in Hex

    printf("proof x mem is contiguous (dec): %ld\n",
           (ptr2 - ptr1)*sizeof(int)); // 9 * 4 = 36 (dec)
                                    // = 24 in Hex

    return 0;
}
```

```
whovian@phy504:~/host/C$ ./ptrarith2
x[4] = 8
proof x mem is contiguous: 0x24
proof x mem is contiguous (dec): 36
```

Pointer arithmetic is super useful to speed-up access to elements stored on contiguous containers

Source:

<https://www.geeksforgeeks.org/pointer-arithmetics-in-c-with-examples/>

# Function Pointers

```
#include <stdio.h>

// Source:
// www.geeksforgeeks.org/function-pointer-in-c/

void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* above line is equivalent of following two
    void (*fun_ptr)(int);
    fun_ptr = &fun;
    */

    (*fun_ptr)(10); // Invoking fun() using fun_ptr
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc pointers19.c -o pointers19
whovian@phy504:~/host/C$ ./pointers19
Value of a is 10
```

- Like normal data pointers, we can have pointers to functions
- Unlike normal pointers, we do not allocate or deallocate memory using function pointers.
- Unlike normal pointers, a function pointer points to code, not data. Function pointer stores the start of executable code.

Source:

<https://www.geeksforgeeks.org/function-pointer-in-c/>

```
#include <stdio.h>

void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}

void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}

void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{ // array of function pointers
    void (*fun_ptr_arr[])(int, int) =
        {add, subtract, multiply};

    int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for "
           "subtract and 2 for multiply\n");

    scanf("%i", &ch);
    if (ch > 2)
    {
        return 0;
    }
    (*fun_ptr_arr[ch])(a, b);
    return 0;
}
```

# Function Pointers

Like normal pointers, we can have an array of function pointers.

Source: <https://www.geeksforgeeks.org/function-pointer-in-c/>

```
#include <stdio.h>

void fun1()
{
    printf("Fun1\n");
}

void fun2()
{
    printf("Fun2\n");
}

// A function that receives a simple func
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}

int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
}
```

# Function Pointers

Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

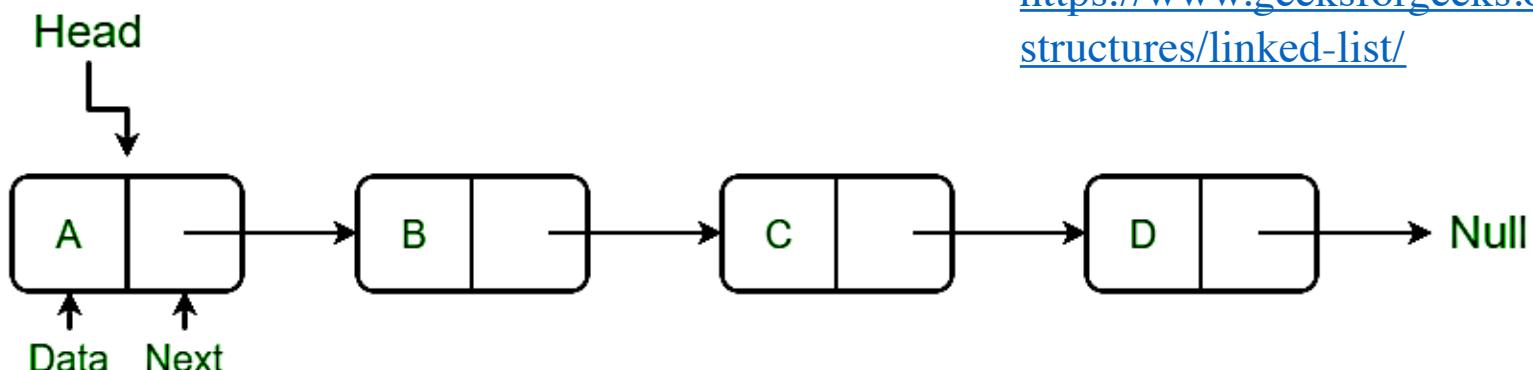
Source:

<https://www.geeksforgeeks.org/function-pointer-in-c/>

# Linked List

Linked list is a linear data structure in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers

```
// I wrote Node on the top as well so I can  
// add a pointer to the same type (the next  
// pointer). Without this, the compiler would have  
// been lost  
typedef struct Node  
{ // A linked list node  
    int data;  
    // Here we need to use the word struct even  
    // with typedef  
    struct Node* next;  
} Node;
```



Source:

<https://www.geeksforgeeks.org/data-structures/linked-list/>

# Linked List

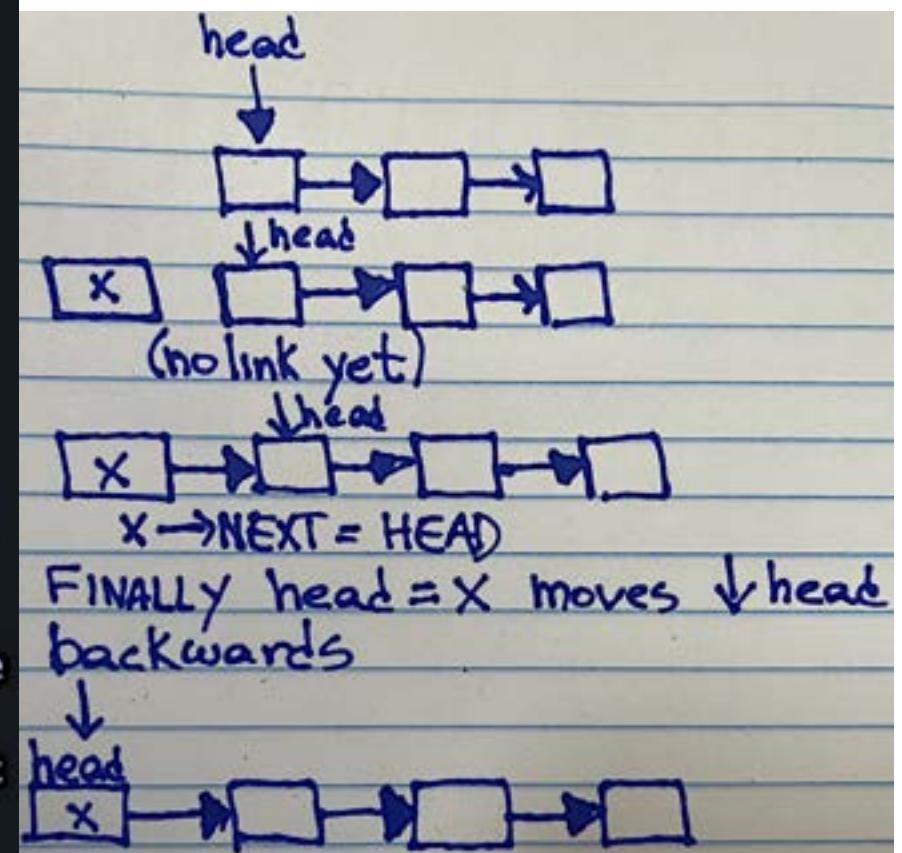
## Adding element to the top of the LL

```
void push(Node** head_ref, int data)
{
    // 1. allocate node
    // Remember out class on pointers
    // Without malloc, Node x would be erased
    // when the function stack collapses.
    Node* x = (Node*) malloc(sizeof(Node));

    x->data = data; // 2. put in the data

    // 3. Next of x links to the original head
    x->next = (*head_ref);

    // 4. move the head to point to the new node
    // This is why we need pointer to pointer
    // We are modifying the original linked list
    (*head_ref) = x;
}
```



Source: <https://www.geeksforgeeks.org/data-structures/linked-list/>

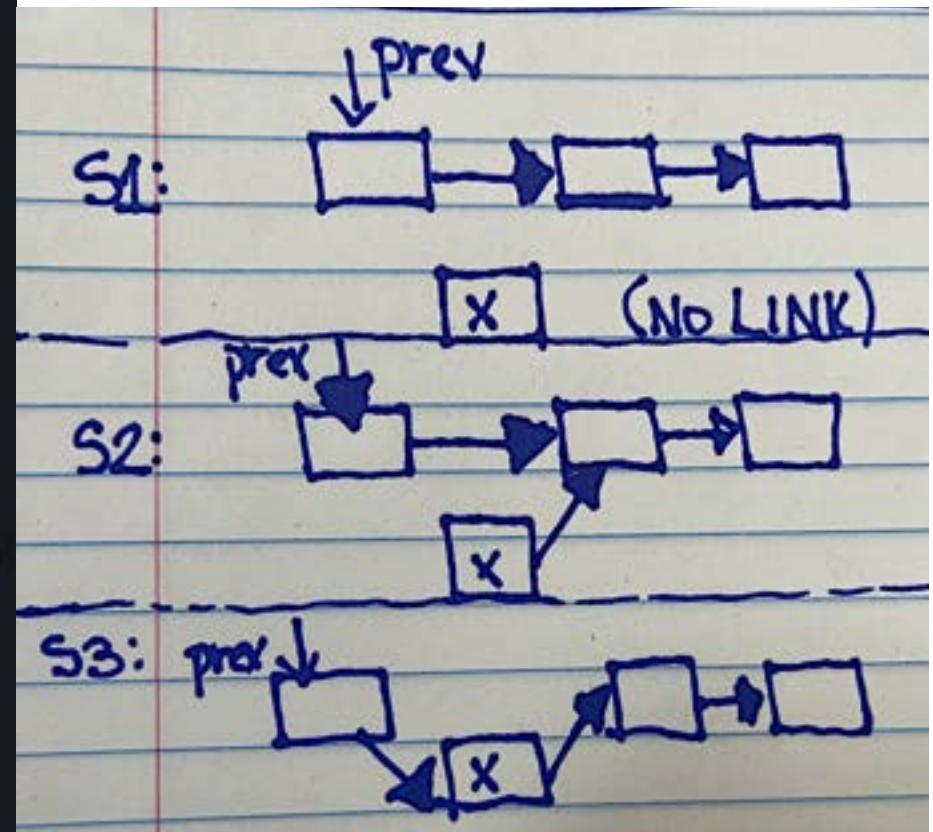
# Linked List

Add a node after a given node

```
void InsertAfter(Node* prev_node, int data)
{
    if (prev_node == NULL)
    {
        printf("node cannot be NULL");
        return;
    }
    // allocate new node & write data
    Node* x = (Node*) malloc(sizeof(Node));
    x->data = data;

    // Make next of new node as next of prev_node
    x->next = prev_node->next;

    // move the next of prev_node as new node
    prev_node->next = x;
}
```



Source: <https://www.geeksforgeeks.org/data-structures/linked-list/>

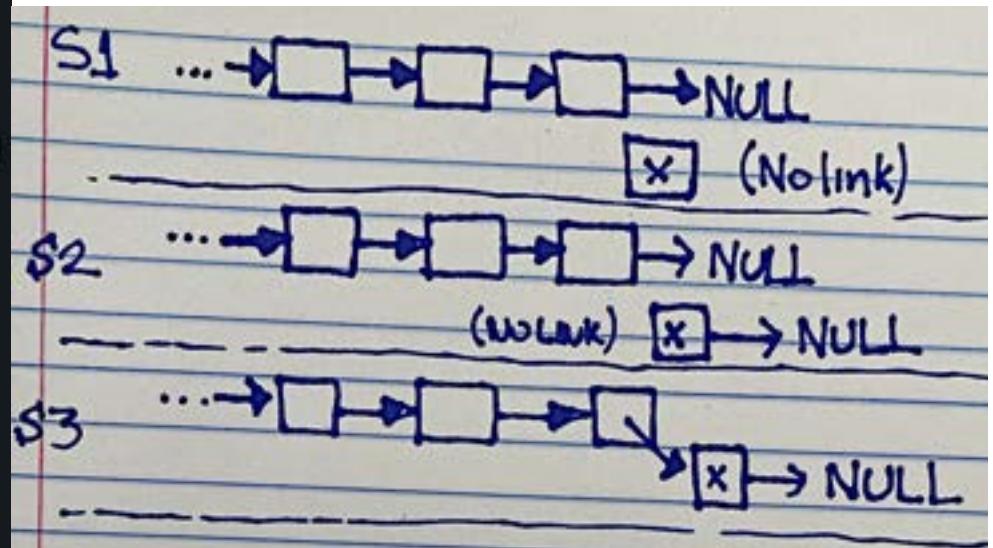
# Linked List

## Add a node at the end

```
void append(Node** head_ref, int data)
{
    // allocate node & write data to new node
    Node* x = (Node*) malloc(sizeof(Node));
    x->data = data;
    x->next = NULL; // this will be the last node

    if (*head_ref == NULL)
    { // if LL is empty, then make the new node as head
        *head_ref = x; // we modify the pointer that is
                        // why the pointer to pointer arg
        return;
    }
    else
    {
        Node* last = *head_ref;

        while (last->next != NULL)
        { // traverse till the last node
            last = last->next;
        }
        last->next = x; // Change the next of last node
    }
}
```

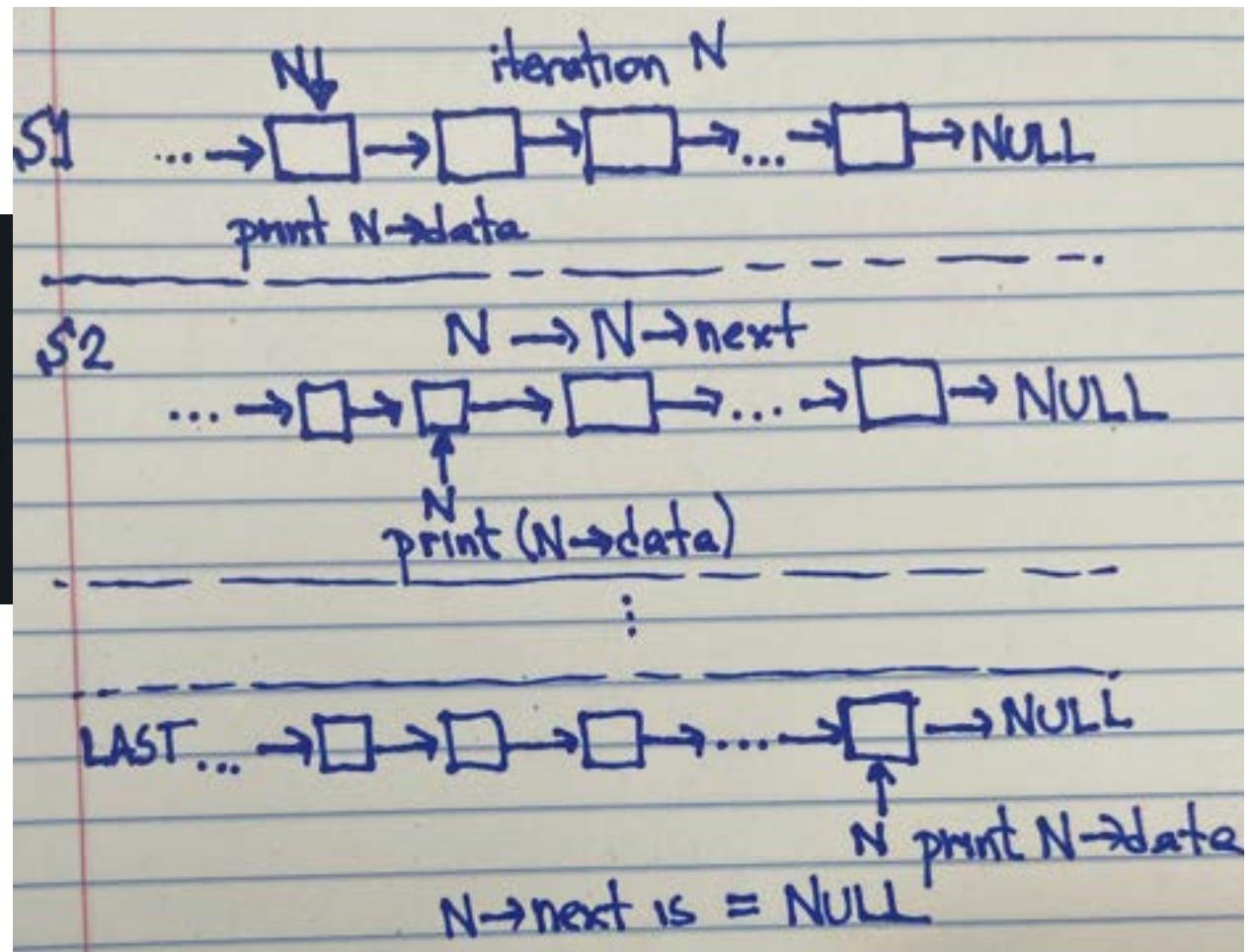


Source: <https://www.geeksforgeeks.org/data-structures/linked-list/>

# Linked List

Showing all the functions that add elements in practice

```
void PrintList(Node* node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
```



# Linked List

Showing all the functions that add elements in practice

```
int main()
{
    // Start with the empty list
    Node* head = NULL;

    append(&head, 6); // linked list becomes 6->NULL
    push(&head, 7); // linked list becomes 7->6->NULL
    push(&head, 1); // linked list becomes 1->7->6->NULL
    append(&head, 4); // linked list becomes 1->7->6->4->NULL

    // linked list becomes 1->7->8->6->4->NULL
    // Remember that head->data = 1
    // head->next->data = 8
    InsertAfter(head->next, 8);

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc linkedlist1.c -o linkedlist1
whovian@phy504:~/host/C$ ./linkedlist1
Created Linked list is: 1 7 8 6 4
```

# Linked List

How to count the number of elements in a linked list

```
int GetCount(Node* head)
{
    int count = 0;          // Initialize count
    Node* current = head; // Initialize current

    while (current != NULL)
    {
        count++;
        current = current->next;
    }
    return count;
}
```

Because memory is non-contiguous, we can't access it via pointer arithmetic. To count elements, we need to visit all nodes until we hit the **NULL** pointer,

Source:

<https://www.geeksforgeeks.org/data-structures/linked-list/>

# Linked List

```
bool search(Node* head, int x)
{
    Node* current = head; // Init current

    while (current != NULL)
    {
        if (current->data == x)
        {
            return true;
        }
        current = current->next;
    }
    return false;
}
```

Checks whether the value x is present in linked list

Because memory is non-contiguous, we need to visit all nodes to check if element exists

No pointer arithmetic possible to get where the next element will be

Source:

<https://www.geeksforgeeks.org/data-structures/linked-list/>

# Linked List

```
void ltreverse(Node** head_ref)
{
    Node* prev = NULL;
    Node* current = *head_ref;
    Node* next = NULL;

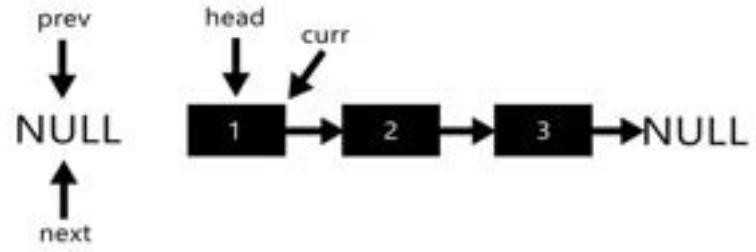
    while (current != NULL)
    {
        next = current->next; // store next
        // Reverse current node's pointer.
        // instead of pointing to the next node,
        // current will point to the prev node.
        // Next is not lost because we stored it.
        current->next = prev;

        // Move pointers one position ahead.
        prev = current;
        current = next;
    }
    *head_ref = prev; // current is null
}
```

How to reverse a linked list?

Source:

<https://www.geeksforgeeks.org/reverse-a-linked-list/>



```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

# Linked List

```
int main()
{
    // Start with the empty list
    Node* head = NULL;

    append(&head, 6); // linked list becomes 6->NULL
    push(&head, 7); // linked list becomes 7->6->NULL
    push(&head, 1); // linked list becomes 1->7->6->NULL
    append(&head, 4); // linked list becomes 1->7->6->4->NULL

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");

    // now reverse it
    ltreverse(&head);
    printf("Reversed Linked list is: ");
    PrintList(head);
    printf("\n");

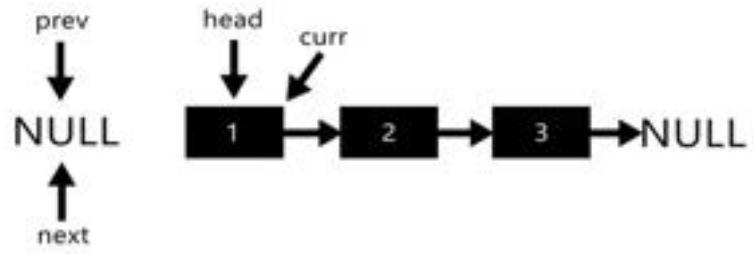
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc linkedlist3.c -o linkedlist3
whovian@phy504:~/host/C$ ./linkedlist3
Created Linked list is: 1 7 6 4
Reversed Linked list is: 4 6 7 1
```

## How to reverse a linked list?

Source:

<https://www.geeksforgeeks.org/reverse-a-linked-list/>

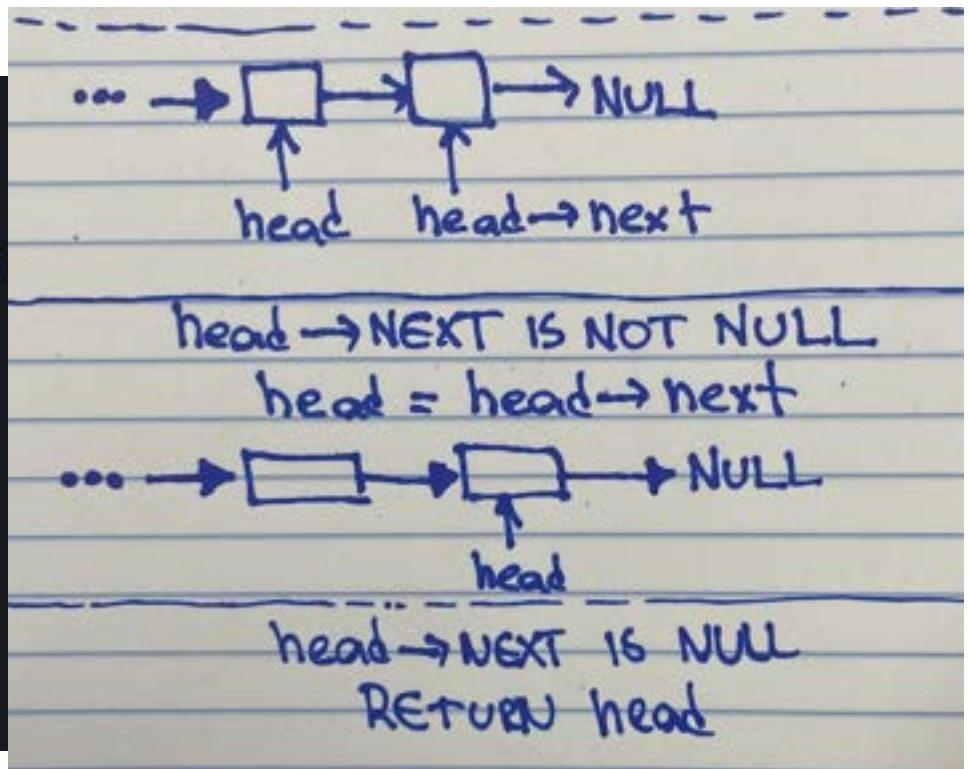


```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

# Linked Lists

How to find last element a linked list?

```
Node* LastNode(Node* head)
{ // Finding last node of linked list
    if(head == NULL)
    { // in case the linked list is empty
        return head;
    }
    while (head->next != NULL)
    {
        head = head->next;
    }
    return head;
}
```



Source:

<https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/>

# Linked Lists

How swap consecutive elements on a linked list?

```
void swap(Node* a, Node* b)
{
    int temp = a->data;
    a->data = b->data;
    b->data = temp;
}
```

Source:

<https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/>

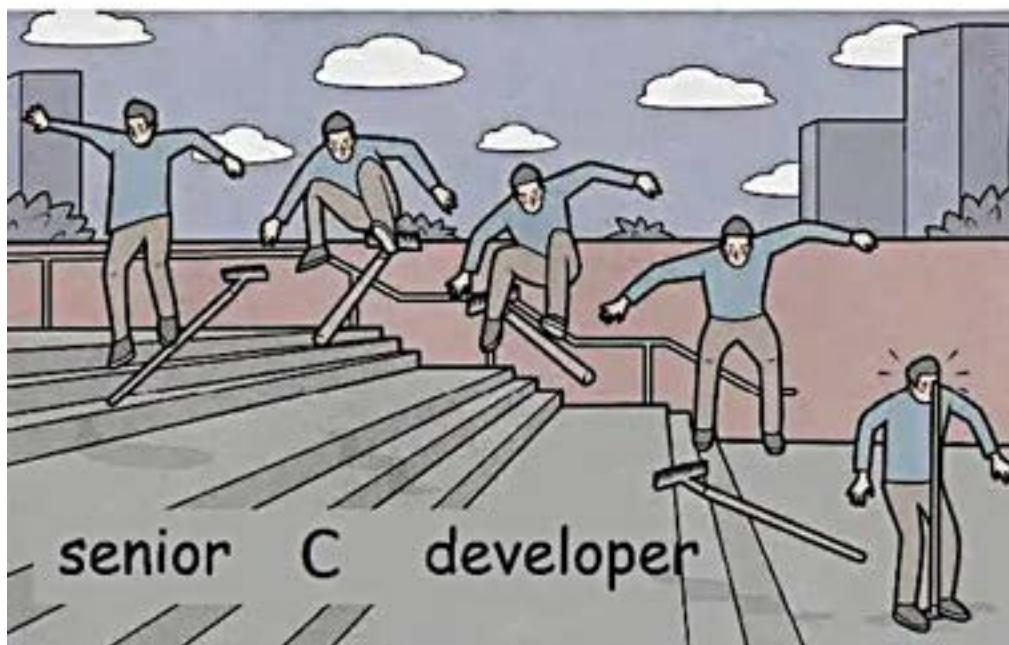
# Lecture 17: C part VIII



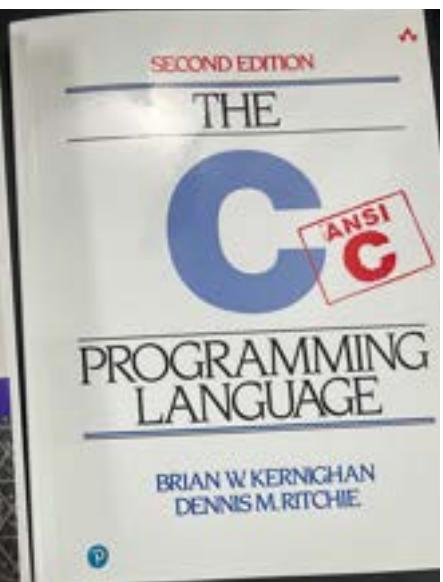
junior C developer



Suggested Literature



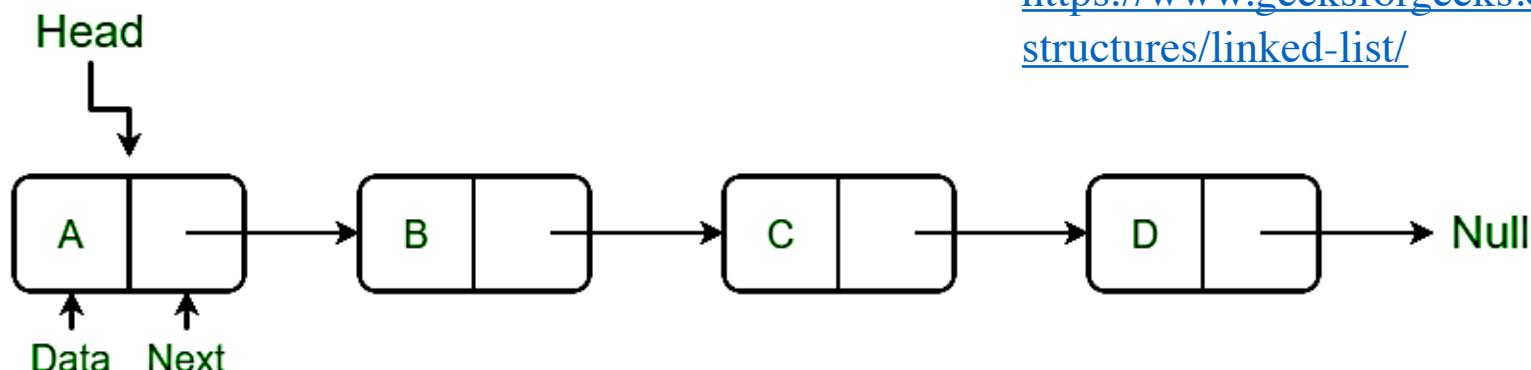
senior C developer



# Linked Lists Part II

Linked list is a linear data structure in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers

```
// I wrote Node on the top as well so I can
// add a pointer to the same type (the next
// pointer). Without this, the compiler would have
// been lost
typedef struct Node
{ // A linked list node
    int data;
    // Here we need to use the word struct even
    // with typedef
    struct Node* next;
} Node;
```



Source:

<https://www.geeksforgeeks.org/data-structures/linked-list/>

# Linked List

```
void ltreverse(Node** head_ref)
{
    Node* prev = NULL;
    Node* current = *head_ref;
    Node* next = NULL;

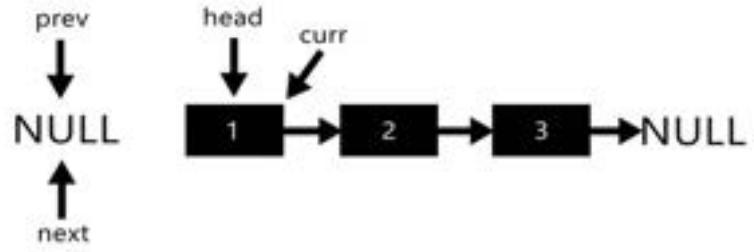
    while (current != NULL)
    {
        next = current->next; // store next
        // Reverse current node's pointer.
        // instead of pointing to the next node,
        // current will point to the prev node.
        // Next is not lost because we stored it.
        current->next = prev;

        // Move pointers one position ahead.
        prev = current;
        current = next;
    }
    *head_ref = prev; // current is null
}
```

How to reverse a linked list?

Source:

<https://www.geeksforgeeks.org/reverse-a-linked-list/>



```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

# Linked List

```
int main()
{
    // Start with the empty list
    Node* head = NULL;

    append(&head, 6); // linked list becomes 6->NULL
    push(&head, 7); // linked list becomes 7->6->NULL
    push(&head, 1); // linked list becomes 1->7->6->NULL
    append(&head, 4); // linked list becomes 1->7->6->4->NULL

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");

    // now reverse it
    ltreverse(&head);
    printf("Reversed Linked list is: ");
    PrintList(head);
    printf("\n");

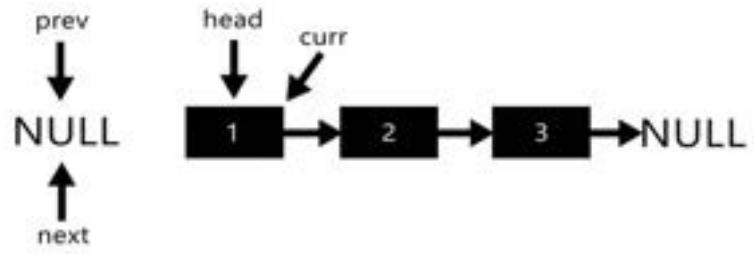
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc linkedlist3.c -o linkedlist3
whovian@phy504:~/host/C$ ./linkedlist3
Created Linked list is: 1 7 6 4
Reversed Linked list is: 4 6 7 1
```

## How to reverse a linked list?

Source:

<https://www.geeksforgeeks.org/reverse-a-linked-list/>



```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

```

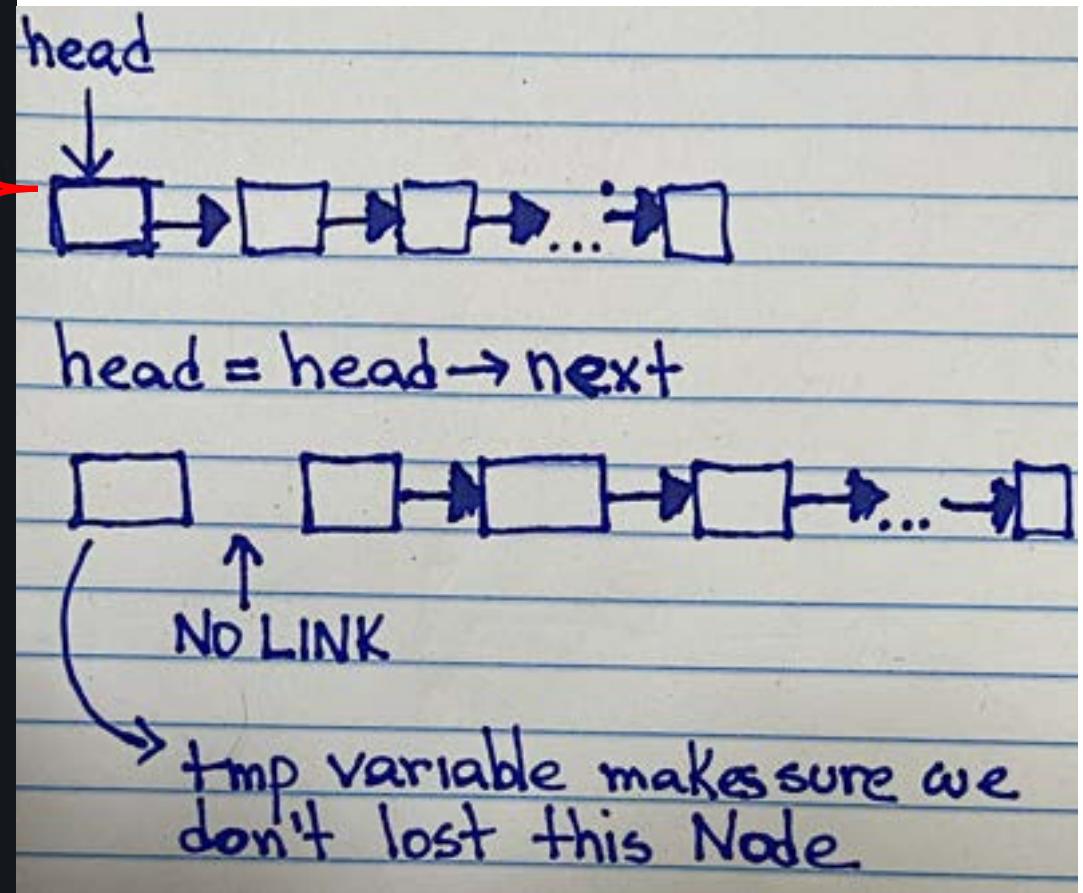
void ltedelete(Node** head_ref, int pos)
{
    assert(pos > 0); // post = 1, first pos
    pos--;
    Node* tmp = *head_ref;
    Node* prev = *head_ref;

    if (pos == 0)
    { // Del first element
        if(tmp != NULL)
        { // if num nodes == 0, nothing to del
            (*head_ref) = (*head_ref)->next;
            free(tmp);
        }
    }
    else
    { // pos > 1
        for (int i=0; i<pos; i++)
        {
            if (tmp == NULL)
            { // in case pos > number of nodes
                break;
            }
            prev = tmp;
            tmp = tmp->next;
        }
        if (tmp != NULL)
        {
            prev->next = tmp->next;
            free(tmp);
        }
    }
}

```

# Linked List

Delete nodes in a linked tree?



Source: <https://www.geeksforgeeks.org/deletion-in-linked-list/>

```

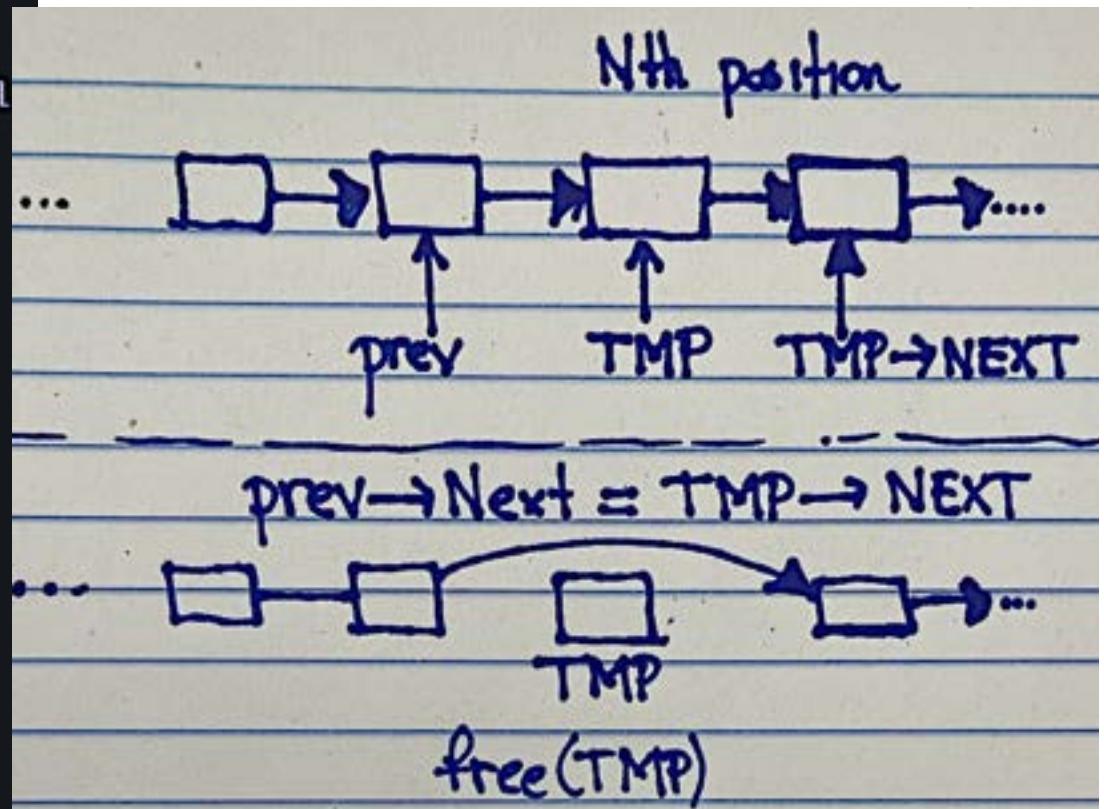
void ltedelete(Node** head_ref, int pos)
{
    assert(pos > 0); // post = 1, first pos
    pos--;
    Node* tmp = *head_ref;
    Node* prev = *head_ref;

    if (pos == 0)
    { // Del first element
        if(tmp != NULL)
        { // if num nodes == 0, nothing to del
            (*head_ref) = (*head_ref)->next;
            free(tmp);
        }
    }
    else
    { // pos > 1
        for (int i=0; i<pos; i++)
        {
            if (tmp == NULL)
            { // in case pos > number of nodes
                break;
            }
            prev = tmp;
            tmp = tmp->next;
        }
        if (tmp != NULL)
        {
            prev->next = tmp->next;
            free(tmp);
        }
    }
}

```

# Linked List

Delete nodes in a linked tree?



Source: <https://www.geeksforgeeks.org/deletion-in-linked-list/>

```
int main()
{
    Node* head = NULL;

    append(&head, 1);
    push(&head, 2);
    push(&head, 3);
    push(&head, 4);
    push(&head, 5);
    push(&head, 6);
    push(&head, 7);

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");

    // now delete first element
    ltdelete(&head, 1);
    PrintList(head);
    printf("\n");

    // now delete third element
    ltdelete(&head, 3);
    PrintList(head);
    printf("\n");

    return 0;
}
```

# Linked List

How to delete nodes in a linked tree?

Source: <https://www.geeksforgeeks.org/deletion-in-linked-list/>

```
whovian@phy504:~/host/C$ ./linkedlist6
Created Linked list is: 7 6 5 4 3 2 1
                           6 5 4 3 2 1
                           6 5 3 2 1
```

```

void RemoveDuplicates(Node* head)
{
    if (head == NULL)
    { // do nothing if the list is empty
        return;
    }
    while (head->next != NULL)
    {
        if (head->data == head->next->data)
        {
            Node* tmp = head->next;
            head->next = head->next->next;
            free(tmp);
        }
        else
        {
            head = head->next;
        }
    }
}

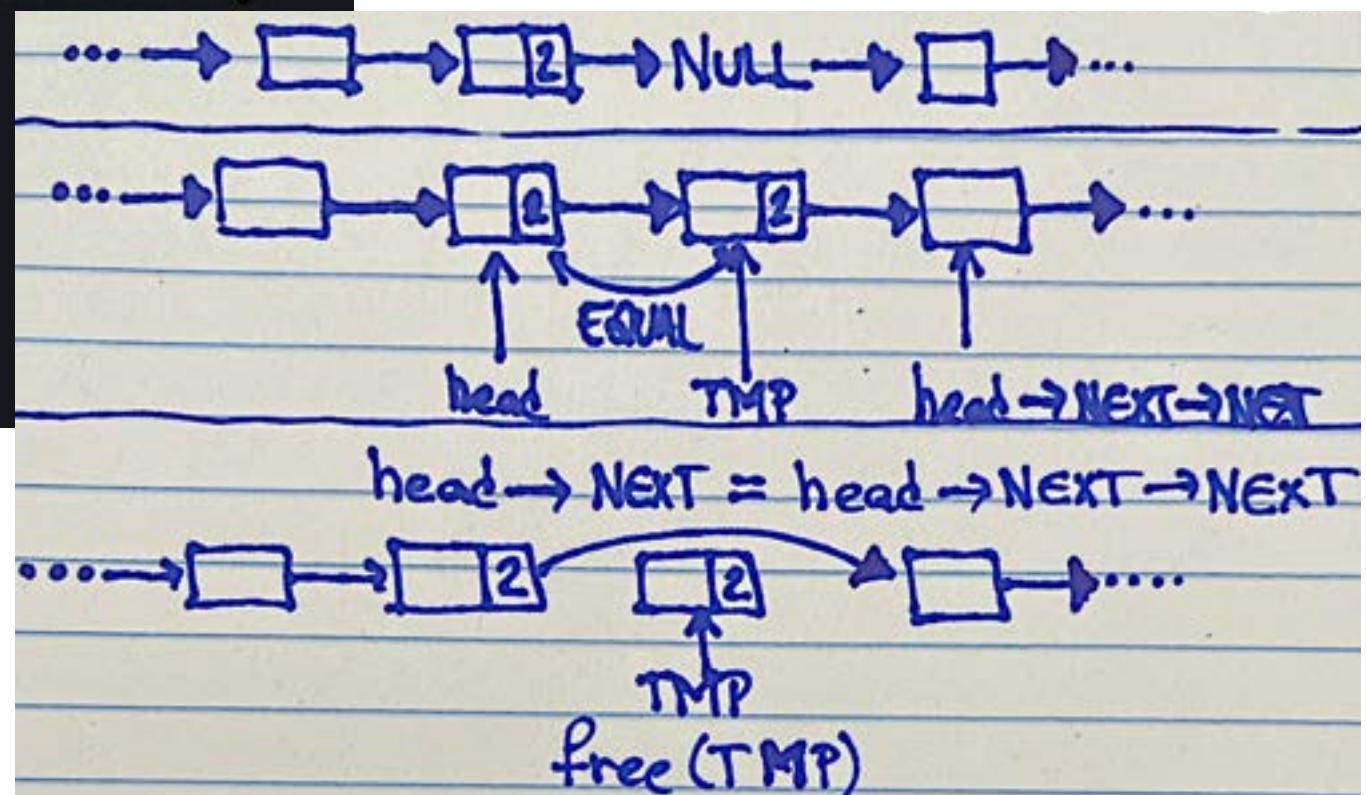
```

# Linked List

How to remove duplicates in a sorted linked tree?

Source:

<https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/>



```
int main()
{
    Node* head = NULL;

    append(&head, 6);
    push(&head, 20);
    push(&head, 13);
    push(&head, 13);
    push(&head, 11);
    push(&head, 11);
    push(&head, 11);
    push(&head, 10);
    push(&head, 9);

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");

    // now delete duplicates
    RemoveDuplicates(head);
    printf("Linked list w/ no duplicates: ");
    PrintList(head);
    printf("\n");

    return 0;
}
```

# Linked List

How to remove duplicates  
in a sorted linked tree?

Source:

<https://www.geeksforgeeks.org/deletion-in-linked-list/>

```
whovian@phy504:~/host/C$ gcc linkedlist7.c -o linkedlist7
whovian@phy504:~/host/C$ ./linkedlist7
Created Linked list is: 9 10 11 11 11 13 13 20 6
Linked list w/ no duplicates: 9 10 11 13 20 6
```

# Linked Lists

How swap consecutive elements on a linked list?

```
void swap(Node* a, Node* b)
{
    int temp = a->data;
    a->data = b->data;
    b->data = temp;
}
```

Source:

<https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/>

```
void sort(Node* head)
{
    if(head == NULL)
    {
        return;
    }

    int swapped;
    do
    {
        swapped = 0;
        Node* p0 = head;

        while (p0->next != NULL) Pass
        {
            if(p0->data > p0->next->data)
            {
                swap(p0, p0->next);
                swapped = 1;
            }
            p0 = p0->next;
        }
    } while (swapped == 1); whovian@phy504:~/host/C$ ./linkedlist8
}
```

# Sort in Linked List

Repeat “Pass” until an entire iteration without any swaps

$(51428) \rightarrow (15428)$ ,  $(14258) \rightarrow (14258)$   
 $(14258) \rightarrow (12458)$ ,  
 $(15428) \rightarrow (14528)$ ,  $(12458) \rightarrow (12458)$   
 $(14528) \rightarrow (14258)$ ,  $(12458) \rightarrow (12458)$   
 $(14258) \rightarrow (14258)$ ,  $(12458) \rightarrow (12458)$

First Pass

Second Pass

$(12458) \rightarrow (12458)$   
 $(12458) \rightarrow (12458)$   
 $(12458) \rightarrow (12458)$   
 $(12458) \rightarrow (12458)$

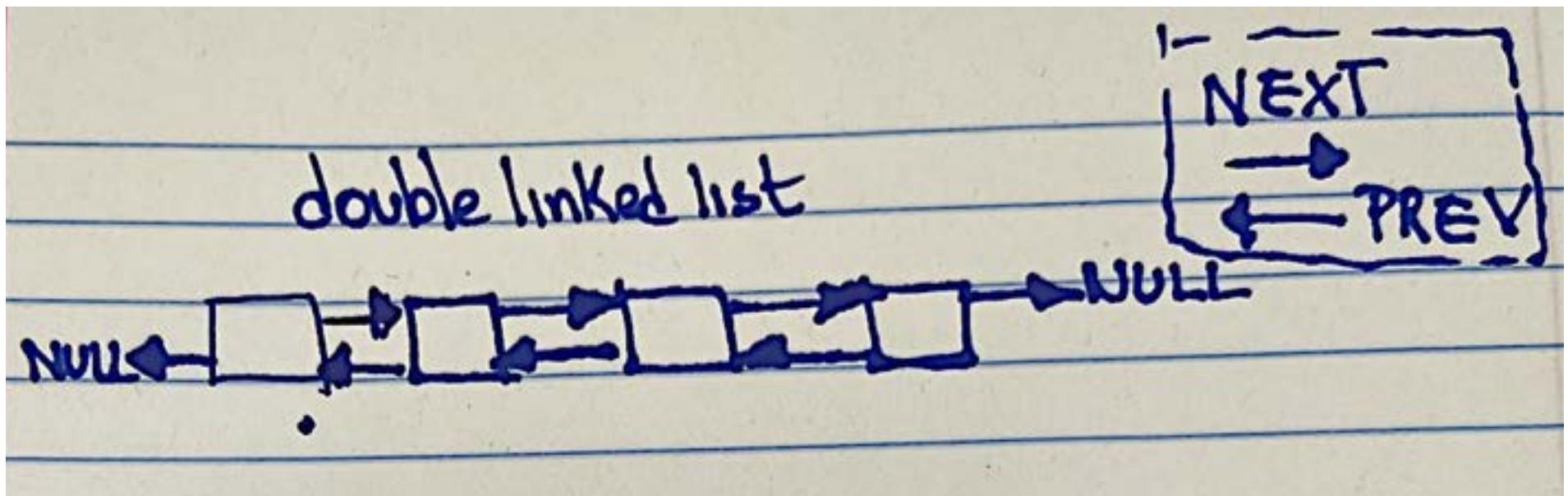
Third Pass

whovian@phy504:~/host/C\$ ./linkedlist8  
Created Linked list is: 6 10 2 5 7 50 3 1  
Linked list w/ no duplicates: 1 2 3 5 6 7 10 50

# Double Linked Lists

Double Linked list is a linear data structure in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers on both directions

Source: <https://www.geeksforgeeks.org/doubly-linked-list-tutorial/>



# Double Linked Lists

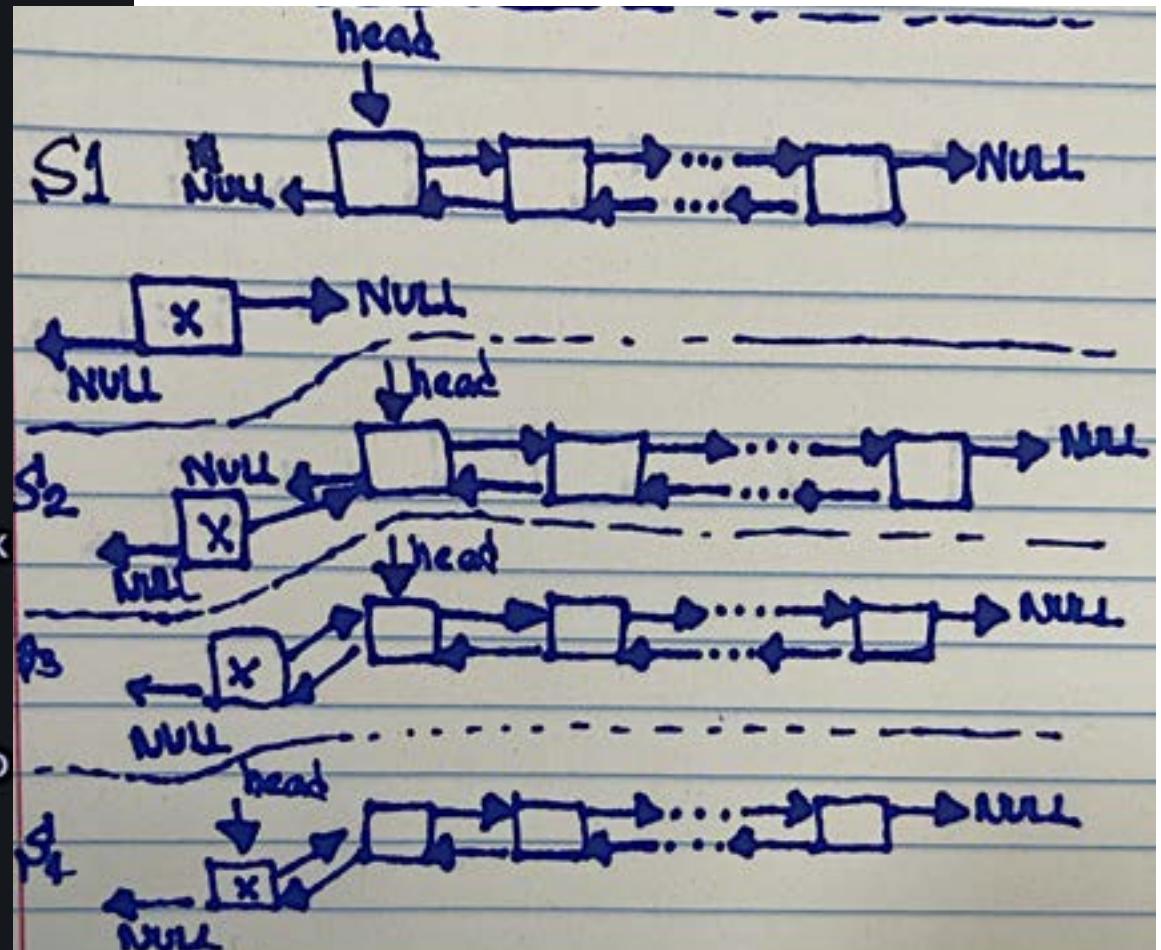
```
void push(Node** head_ref, int data)
{
    //1. create the new node
    Node* x =
        (Node*) malloc(sizeof(Node));
    x->data = data;
    x->next = NULL;
    x->prev = NULL;

    //2. Next of x links to the
    //original head
    x->next = (*head_ref);

    if (*head_ref != NULL)
    {
        //3. prev of head links to x
        (*head_ref)->prev = x;
    }

    //4. move the head to point to
    //the new node
    (*head_ref) = x;
}
```

Adding element to either at the top or end of the DLL



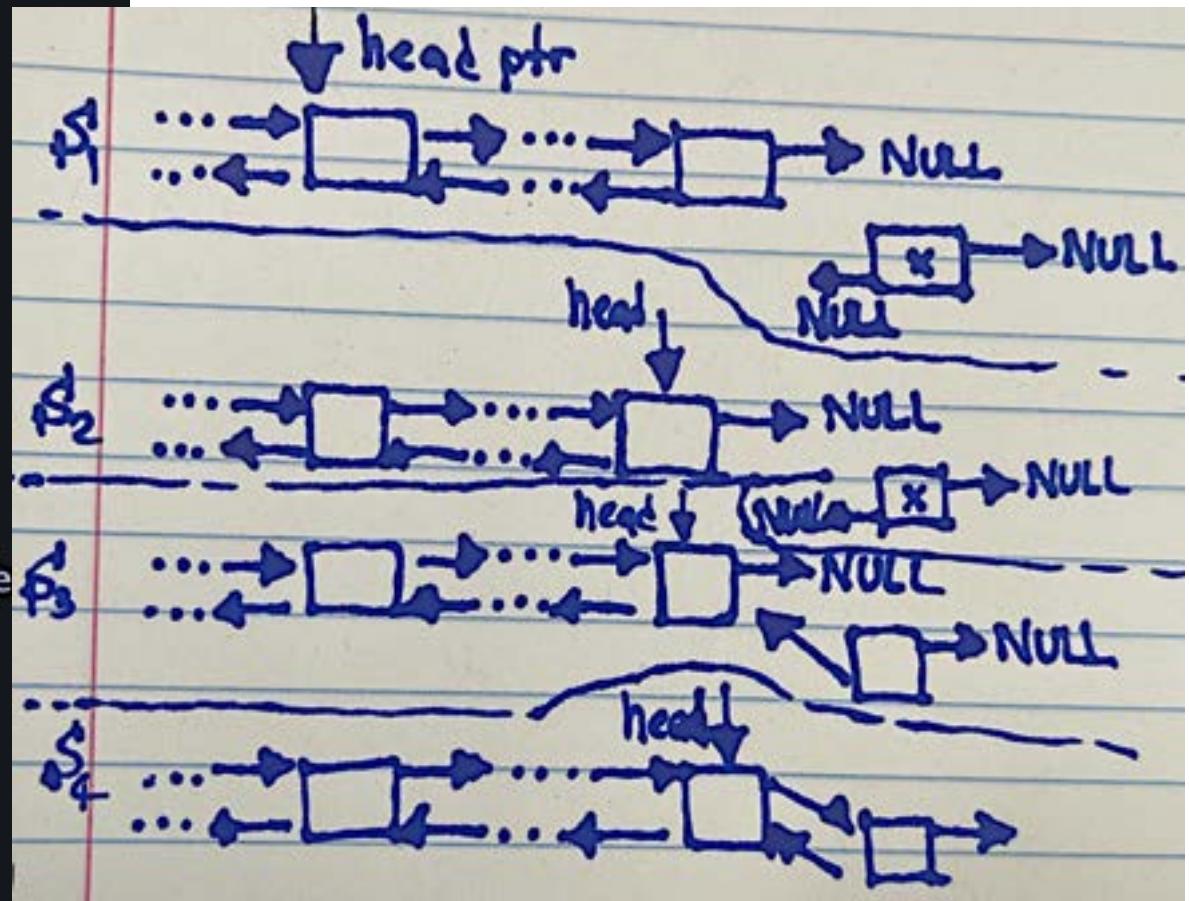
# Double Linked Lists

```
void append(Node** head_ref, int data)
{
    Node* x = (Node*) malloc(sizeof(Node));
    x->data = data;
    x->next = NULL;
    x->prev = NULL;

    if (*head_ref == NULL)
    {
        *head_ref = x;
        return;
    }
    else
    {
        Node* last = *head_ref;

        while (last->next != NULL)
        { // traverse till the last node
            last = last->next;
        }
        last->next = x;
        x->prev = last;
    }
}
```

Adding element to either at the top  
or end of the DLL

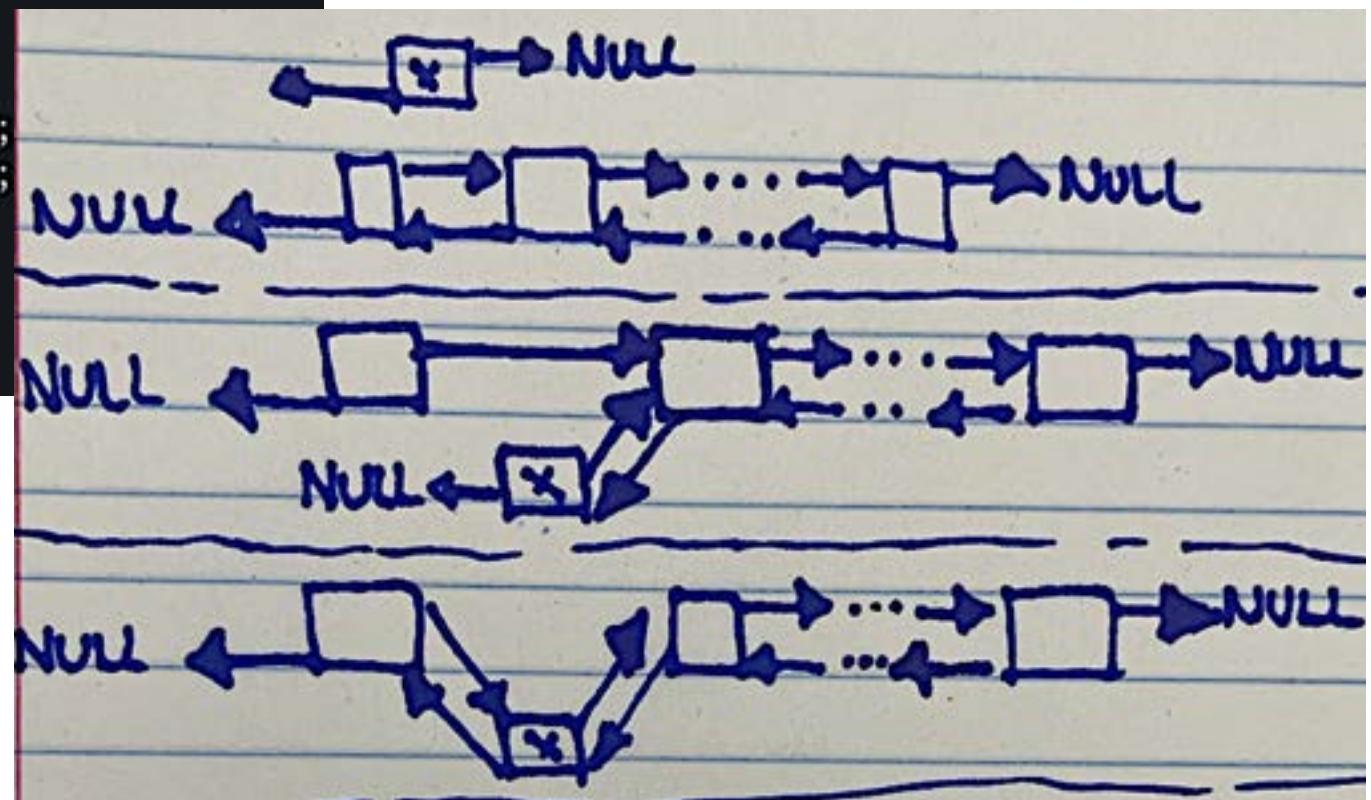


# Double Linked Lists

```
void InsertAfter(Node* prev_node, int data)
{
    if (prev_node == NULL)
    {
        printf("node cannot be NULL");
        return;
    }
    Node* x = (Node*) malloc(sizeof(Node));
    x->data = data;
    x->next = NULL;
    x->prev = NULL;

    x->next = prev_node->next;
    prev_node->next->prev = x;
    prev_node->next = x;
    x->prev = prev_node;
}
```

Add a node after a given node



# Double Linked Lists

Print all nodes with a possible rewind

```
void PrintList(Node* node)
{
    while (node != NULL &&
           node->prev != NULL)
    { // rewind list
        node = node->prev;
    }
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
```

# Double Linked Lists

```
int main()
{
    Node* head = NULL;

    append(&head, 6); // linked list becomes 6->NULL
    push(&head, 7); // linked list becomes 7->6->NULL
    push(&head, 1); // linked list becomes 1->7->6->NULL
    append(&head, 4); // linked list becomes 1->7->6->4->NULL

    // linked list becomes 1->7->8->6->4->NULL
    // Remember that head->data = 1
    //           head->next->data = 8
    InsertAfter(head->next, 8);

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");

    return 0;
}
```

All together

```
whovian@phy504:~/host/C$ gcc dbl1.c -o dbl1
whovian@phy504:~/host/C$ ./dbl1
Created Linked list is: 1 7 8 6 4
```

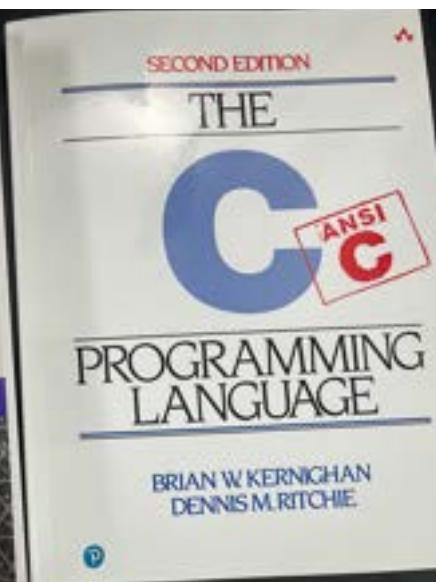
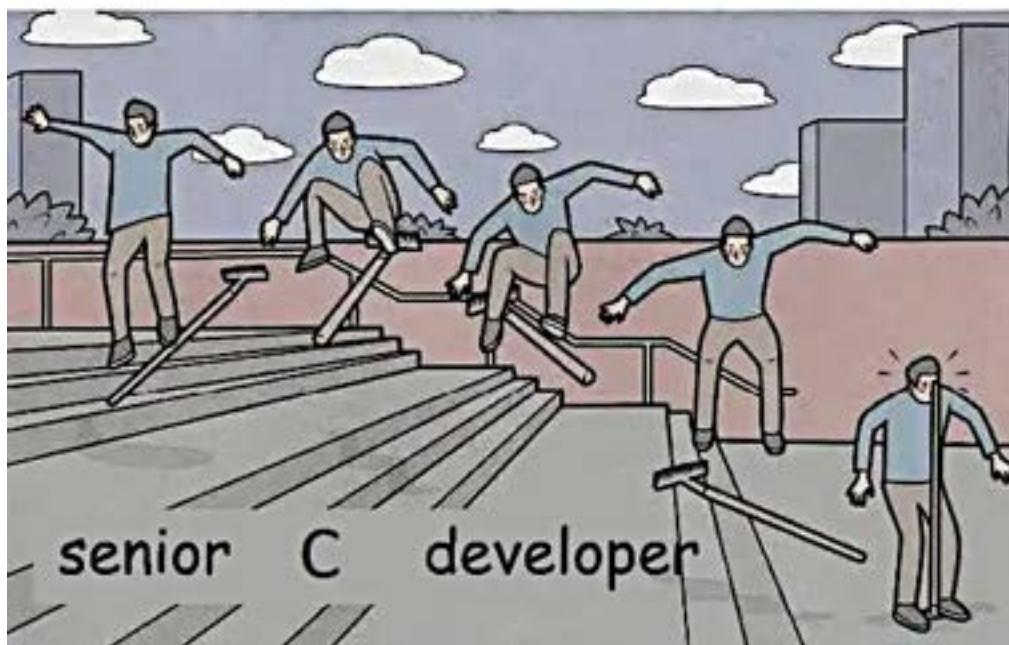
# Lecture 18: C part IX



junior C developer



## Suggested Literature



# Strings in C

Deeper dive on  
**printf** command in  
C: formatting

float numbers

**%e, %f, %g**

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

int main()
{
    const double a = 97.4583;

    // %e and %f default: six decimal places
    printf ("%f %e\n", a, a);

    // %g is weird. Up to 10^-4, %g = %f
    // Lower than 10^-4, %g = %e
    const double b = 0.001;
    const double c = 0.0001;
    const double d = 0.00001;
    printf ("%g %g %g\n", b, c, d);

    // %g is weird. Up to 10^5, %g = %f,
    // Higher than 10^5, %g = %e
    const double e = 10000;
    const double f = 100000;
    const double g = 1000000;
    printf ("%g %g %g\n", e, f, g);

    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc char2.c -o char2
whovian@phy504:~/host/C$ ./char2
97.458300 9.745830e+01
0.001 0.0001 1e-05
10000 100000 1e+06
```

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

int main()
{
    const double a = 97.4583;

    printf ("x=%7.2f\n", a);
    printf ("x=%07.2f\n", a);

    printf ("\n");

    printf ("x=%.2e\n", a);
    printf ("x=%.2e\n", a);

    return 0;
}
```

# Strings in C

Deeper dive on  
**printf** command in  
C: formatting

float numbers

**%e, %f, %g**

```
whovian@phy504:~/host/C$ gcc char3.c -o char3
whovian@phy504:~/host/C$ ./char3
x= 97.46
x=0097.46

x=9.75e+01
x=9.75e+01
```

# Strings in C

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

int main()
{
    const int a = 300;

    printf ("a=%10d\n", a);
    printf ("a=%010d\n", a);

    const int b = 45;
    printf("b=%5x\n", b);
    printf("b=%05X\n", b);

    // no specification for
    // binary :(
    return 0;
}
```

Deeper dive on **printf** command in C:  
formatting

Integer: **%d**  
Hexadecimal: **%x** (0-9a-f) and **%X** (0-9A-F)

```
whovian@phy504:~/host/C$ gcc char5.c -o char5
whovian@phy504:~/host/C$ ./char5
a=      300
a=0000000300
b=      2d
b=0002D
```

# Strings in C

**scanf** in C:  
user defines  
values on  
the terminal

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

int main()
{
    printf("Enter the length of the string: ");

    int n;
    scanf("%d", &n);

    char* str = malloc((n + 1)*sizeof(char));
    if (str == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        return 1;
    }

    printf("Enter the string: ");
    scanf("%s", str);

    printf("The entered string is: %s\n", str);

    free(str);
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc char1.c -o char1
whovian@phy504:~/host/C$ ./char1
Enter the length of the string: 10
Enter the string: Vivian
The entered string is: Vivian
```

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

int main()
{
    const double a = 97.4583;

    int width, prec;
    printf("Enter width\n");
    scanf("%d", &width);
    printf("Enter precision\n");
    scanf("%d", &prec);

    printf ("x=%.*f\n", width, prec, a);
    printf ("x=%0.*f\n", width, prec, a);
    return 0;
}
```

whovian@phy504:~/host/C\$ gcc char4.c -o char4  
whovian@phy504:~/host/C\$ ./char4  
Enter width  
10  
Enter precision  
3  
x= 97.458  
x=000097.458

# Strings in C

What if you want to specify width / precision of float numbers at runtime?

# Strings in C

## Introduction to **char\*** and string literals

**s** and **&s** are the same for **char** arrays

(char array just alias to ptr)

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

int main()
{
    // string literal is weird. Not understanding
    // string literal is dangerous. "string" (with
    // double quotes) is a string literal and cannot
    // be modified. HOWEVER...
    char s[6] = "hello"; // this copy string literal
                          // to the array. So ok to
                          // modify the array

    // We need 6! Why? Because every string literal
    // ends with the '\0' (NULL) character
    printf ("s = %s\n", s); // ←

    // cannot use double quotes here
    // weird: double quotes = string literal (X + '\0')
    //          string literal "X" would
    //          return char*
    //
    // Single quotes = just a char
    s[0] = 'X'; // ok to do it

    printf ("s = %s\n", s);
    return 0;
}
```

```
whovian@phy504:~/host/C$ nano char6.c
whovian@phy504:~/host/C$ gcc -Wall -pedantic char6.c -o char6
whovian@phy504:~/host/C$ ./char6
s = hello
s = Xello
```

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

int main()
{
    // string literal is weird. Not understanding
    // string literal is dangerous. "string" (with
    // double quotes) is a string literal and cannot
    // be modified. HOWEVER...
    char s[6] = "hello"; // this copy string literal
                         // to the array. So ok to
                         // modify the array

    // We need 6! Why? Because every string literal
    // ends with the '\0' (NULL) character
    printf ("s = %s\n", s); ←

    // equivalent statement
    char s2[6] = {'h','e','l','l','o','\0'};

    printf ("s = %s\n", s2);

    return 0;
}
```

# Strings in C

Introduction to  
**char\*** and  
string literals

**s** and **&s** are  
the same for  
**char** arrays

(char array just alias to ptr)

```
whovian@phy504:~/host/C$ nano char7.c
whovian@phy504:~/host/C$ gcc -Wall -pedantic char7.c -o char7
whovian@phy504:~/host/C$ ./char7
s = hello
s = hello
```

# Strings in C

## Introduction to **char\*** and string literals

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

int main()
{
    // string literal is weird. Not understanding
    // string literal is dangerous. "string" (with
    // double quotes) is a string literal and cannot
    // be modified. HOWEVER...
    char s[] = "hello"; // this copy string literal
                        // to the array. So ok to
                        // modify the array

    // When init the array with a string literal,
    // you can use char[]. C will figure out that
    // the size is 6 (remember the null char \0)
    printf ("s = %s. Size = %ld\n", s, sizeof(s));

    // PS: sizeof return long unsigned int (so %ld)
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc -Wall -pedantic char8.c -o char8
whovian@phy504:~/host/C$ ./char8
s = hello. Size = 6
```

# Strings in C

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

int main()
{
    // string literal is weird. Not understanding
    // string literal is dangerous. "string" (with
    // double quotes) is a string literal and cannot
    // be modified. HOWEVER...

    char* s = "hello"; // string literal lives in
                        // read-only part of memory

    // so modifying a char* that points to a string
    // literal is UNDEFINED BEHAVIOR (not impossible
    // because it is not const char*, but it is UNDEFINED
    printf("s = %s\n", s);
    return 0;
}
```



```
whovian@phy504:~/host/C$ nano char9.c
whovian@phy504:~/host/C$ gcc -Wall -pedantic char9.c -o char9
whovian@phy504:~/host/C$ ./char9
s = hello
```

# Strings in C



```
#include <string.h>
int main()
{
    char *str1 = "string Literal";
    const char *str2 = "string Literal";

    char source[] = "Sample string";

    // Why I really support const correctness

    strcpy(str1,source);      // No warning or error, just
                            // Undefined Behavior
    strcpy(str2,source);      //Compiler issues a warning

    return 0;
}
```

The first arg in **strcpy(dest, source)** is the destination. Both arguments in **strcpy** are **char\***

```
whovian@phy504:~/host/C$ nano char11.c
whovian@phy504:~/host/C$ gcc -Wall -pedantic char11.c -o char11
char11.c: In function 'main':
char11.c:13:12: warning: passing argument 1 of 'strcpy' discards 'const' qualifier from pointer target type [-Wdiscarded-qualifiers]
  13 |     strcpy(str2,source);      //Compiler issues a warning
     |             ^
In file included from char11.c:1:
/usr/include/string.h:122:39: note: expected 'char * restrict' but argument is of type 'const char *'
 122 | extern char *strcpy (char * __restrict __dest, const char * __restrict __src)
     |             ^~~~~~
```

# Strings in C

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

// Source: 21st century C. Author: Ben Klemens

int main()
{
    // strip will copy the str literal to a proper char*
    char *str1 = strdup("string Literal is const");
    char source[] = "Sample string";

    strcpy(str1, source); // Not a problem anymore!
    printf("%s\n", str1);
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc char11v2.c -o char11v2
whovian@phy504:~/host/C$ ./char11v2
Sample string
```

An interesting way to avoid this problem is to never assign a string literal to **char\*** directly.

Use **char\* strdup( const char\* s)** to copy the string literal to **char\***

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

int main()
{
    char c;
    printf("Enter some character. Enter $ to exit...\n");

    for(;;)
    {
        c = getchar();
        printf("Entered character is: ");
        if (c == '$')
        {
            break;
        }
        putchar(c);
        printf("\n");
    }

    return 0;
}
```

# Strings in C

## getchar/putchar

read/write a single char from terminal

```
whovian@phy504:~/host/C$ gcc -Wall -pedantic char10.c -o char10
whovian@phy504:~/host/C$ ./char10
Enter some character. Enter $ to exit...
Vivian Miranda$
Entered character is: V
Entered character is: i
Entered character is: v
Entered character is: i
Entered character is: a
Entered character is: n
Entered character is:
Entered character is: M
Entered character is: i
Entered character is: r
Entered character is: a
Entered character is: n
Entered character is: d
Entered character is: a
```

# Strings in C

You can use bash input and output redirection to aid reading and printing strings in C

```
whovian@phy504:~/host$ ./char10
Enter some character. Enter $ to exit...
1234$
Entered character is: 1
Entered character is: 2
Entered character is: 3
Entered character is: 4
Entered character is: whovian@phy504:~/host$ nano inputchar10.txt
whovian@phy504:~/host$ ./char10 < inputchar10.txt
Enter some character. Enter $ to exit...
Entered character is: 1
Entered character is: 2
Entered character is: 3
Entered character is: 4
Entered character is: whovian@phy504:~/host$ more inputchar10.txt
1234$
```

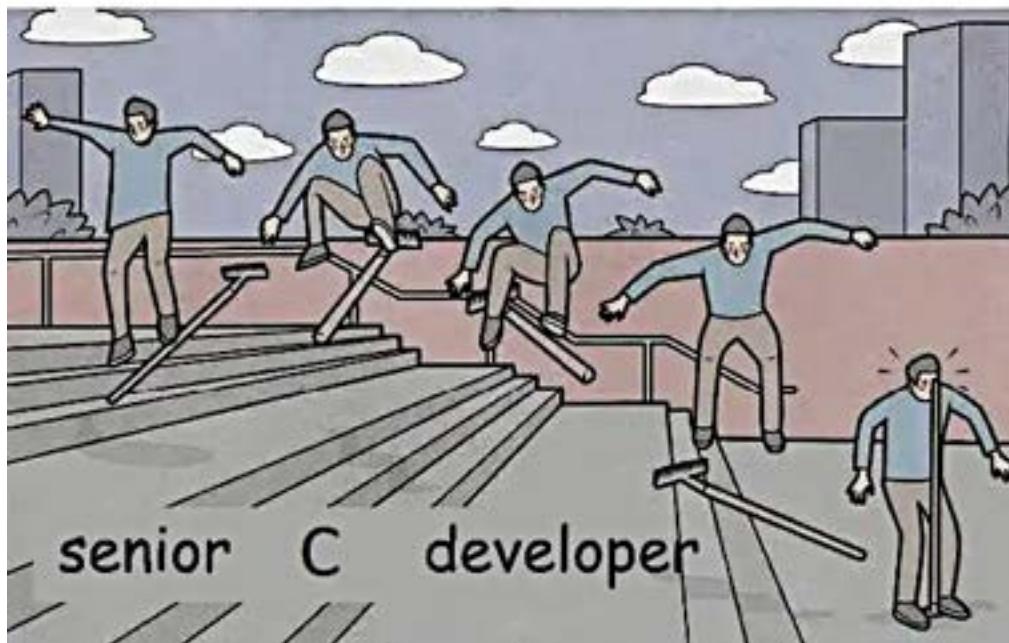
# Lecture 19: C part X



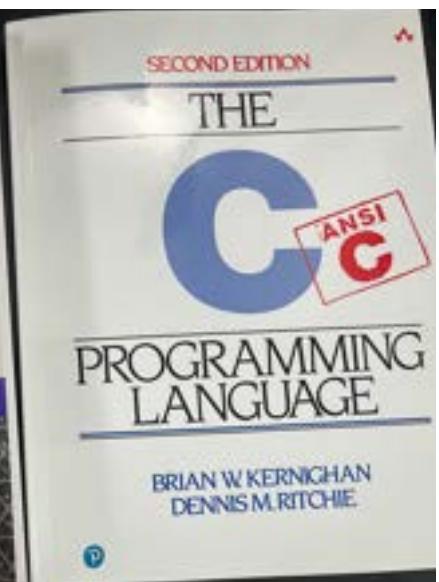
junior C developer



Suggested Literature



senior C developer



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

int main()
{
    char* buffer;
    size_t bufsize = 32; // long unsigned int
    size_t chars;      // long unsigned int

    buffer = (char*) malloc(bufsize*sizeof(char));
    if (buffer == NULL)
    {
        perror("Unable to allocate buffer");
        exit(1);
    }

    printf("Type something: ");
    chars = getline(&buffer, &bufsize, stdin);
    printf("%zu characters were read. \n", chars);
    printf("You type: %s \n", buffer);

    return 0;
}
```

# Strings in C

**getline** allows you to read entire lines (up to **\n**)

Be careful with buffer overflow!

```
whovian@phy504:~/host$ gcc getline.c -o getline
whovian@phy504:~/host$ ./getline
Type something: Interesting
12 characters were read.
You type: Interesting
```

# Strings in C

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

int main()
{
    char* buffer = NULL;
    size_t bufsize = 0;
    size_t chars;

    printf("Type something: ");
    chars = getline(&buffer, &bufsize, stdin);
    printf("%zu characters were read. \n", chars);
    printf("You type: %s \n", buffer);

    return 0;
}
```

if you provide a **NULL** pointer, will allocate sufficient storage for your input

**malloc** only relevant when dealing with multiple inputs (ex: double pointer for multiple lines)

```
whovian@phy504:~/host$ gcc getline2.c -o getline2
whovian@phy504:~/host$ ./getline2
Type something: Vivian Miranda
15 characters were read.
You type: Vivian Miranda

whovian@phy504:~/host$ █
```

# Strings in C

You can use bash input and output redirection to aid reading and printing strings in C

```
whovian@phy504:~/host$ gcc char1.c -o char1
whovian@phy504:~/host$ ./char1
Enter the length of the string: 10
Enter the string: Vivian
The entered string is: Vivian
whovian@phy504:~/host$ more inputchar1.txt
10
Vivian
whovian@phy504:~/host$ ./char1 < inputchar1.txt
Enter the length of the string: Enter the string: The entered string is: Vivian
```

```
whovian@phy504:~/host$ gcc helloworld.c -o hello
whovian@phy504:~/host$ ./hello
Physics 504 is fun.
whovian@phy504:~/host$ ./hello > hello.txt
whovian@phy504:~/host$ more hello.txt
Physics 504 is fun.
```

# Strings in C

You can use bash input and output redirection to aid reading and printing strings in C

```
whovian@phy504:~/host$ gcc getline.c -o getline
whovian@phy504:~/host$ ./getline
Type something: Vivian Miranda
15 characters were read.
You type: Vivian Miranda
```

```
whovian@phy504:~/host$ more getlineinput.txt
Vivian Miranda
```

```
whovian@phy504:~/host$ ./getline < getlineinput.txt
Type something: 15 characters were read.
You type: Vivian Miranda
```

```
whovian@phy504:~/host$ █
```

# Working with files

```
#include <stdio.h>

int main (void)
{
    int c;
    while ((c = getchar ()) != EOF)
    {
        putchar (c);
    }
    return 0;
}
```

C has a special keyword to denote **end-of-file**: **EOF**

```
whovian@phy504:~/host$ gcc eof.c -o eof
whovian@phy504:~/host$ nano eofinput.txt
whovian@phy504:~/host$ ./eof < eofinput.txt
My name is Vivian
I live in NYC
I was born in Brazil
whovian@phy504:~/host$ more eofinput.txt
My name is Vivian
I live in NYC
I was born in Brazil
```

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

// reverse string, swaps src & dest each iteration up to '\n'
// Code source: https://stackoverflow.com/a/66132517/2472169
char* strrev (char* str)
{
    // original string is not preserved
    if (!str)
    {
        // validate str not NULL
        printf ("error: invalid string\n");
        return NULL;
    }
    if (!*str)
    {
        // check empty string
        return str;
    }
    char* begin = str;
    char* end = begin + strcspn (str, "\n") - 1;

    while (end > begin)
    {
        char tmp = *end;
        *end-- = *begin;
        *begin++ = tmp;
    }
    return str;
}
int main()
{
    char* buffer = NULL;
    size_t bufsize = 0;
    size_t chars;
    chars = getline(&buffer, &bufsize, stdin);
    printf("You type in reverse: %s \n", strrev(buffer));
    return 0;
}
```

# Strings in C

We can use **strrev** to reverse lines read via **getline** (or **scanf**)

## strcspan(str1, str2)

returns the length of the initial part of **str1** **not** containing any of the characters that are part of **str2**

There are many useful functions on **string.h** header

```
whovian@phy504:~/host$ gcc getline3.c -o getline3
whovian@phy504:~/host$ ./getline3
Vivian Miranda
You type in reverse: adnariM naiviV
```

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

// reverse string, swaps src & dest each iteration up to '\n'
// Code source: https://stackoverflow.com/a/66132517/2472169
char* strrev (char* str)
{
    // original string is not preserved
    if (!str)
    {
        // validate str not NULL
        printf ("error: invalid string\n");
        return NULL;
    }
    if (!*str)
    {
        // check empty string
        return str;
    }
    char* begin = str;
    char* end = begin + strcspn (str, "\n") - 1;

    while (end > begin)
    {
        char tmp = *end;
        *end-- = *begin;
        *begin++ = tmp;
    }
    return str;
}

int main()
{
    char* buffer = NULL;
    size_t bufsize = 0;
    while (getline(&buffer, &bufsize, stdin) != EOF)
    {
        printf("You type in reverse: %s \n", strrev(buffer));
    }
    return 0;
}
```

# Strings in C

The use of **strcspan** is quite important when dealing with multiple lines (we want to reverse them separately)

```
whovian@phy504:~/host$ gcc getline4.c -o getline4
whovian@phy504:~/host$ ./getline4 < eofinput.txt
You type in reverse: naiviV si eman yM
You type in reverse: CYN ni evil I
You type in reverse: lizarB ni nrob saw I
whovian@phy504:~/host$ █
```

```
#include <stdio.h>
#include <string.h>
// source: https://www.geeksforgeeks.org/strpbrk-in-c/
int main()

{
    char* str1 = "geeksforgeeks";
    char* str2 = "app";
    char* str3 = "kite";

    char* r = strpbrk(str1, str2);
    if (r != 0)
    {
        printf("First matching character: %c\n", *r);
    }
    else
    {
        printf("Character not found\n");
    }

    char* t = strpbrk(str1, str3);
    if (t != 0)
    {
        printf("First matching character: %c\n", *t);
    }
    else
    {
        printf("Character not found\n");
    }
    return 0;
}
```

# Strings in C

There are useful functions on **String.h**

**strpbrk** finds the first character in the string **str1** that matches any character specified in **str2**

```
whovian@phy504:~/host$ gcc char12.c -o char12
whovian@phy504:~/host$ ./char12
Character not found
First matching character: e
```

# Strings in C

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

// source: https://www.geeksforgeeks.org/strstr-in-cpp/

int main()
{
    char* s1 = "GeeksforGeeks";
    char* s2 = "for";
    char* p = strstr(s1, s2);

    if (p != NULL) // equivalent to if(p)
    {
        printf("String found\n");
        printf("First occurrence of string '%s' in '%s' is '%s' \n",
               s2, s1, p);
    }
    else
    {
        printf("String not found\n");
    }
    return 0;
}
```

whovian@phy504:~/host\$ gcc char13.c -o char13  
whovian@phy504:~/host\$ ./char13  
String found  
First occurrence of string 'for' in 'GeeksforGeeks' is 'forGeeks'

What if we want to find a string?

**strstr** returns a pointer to the first occurrence of **string str2** in **str1**

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

// char *strncat(char* dest,
//                 const char* src)

// returns a pointer to the
// resulting string dest.

int main ()
{
    char str[80];
    strcpy (str,"these ");
    strcat (str,"strings ");
    strcat (str,"are ");
    strcat (str,"concatenated.");

    printf("Resulting string = \n %s\n", str);

    return 0;
}
```

# Strings in C

How to concatenate  
two strings?

Two options: **strcat**  
or **strncat**

Pointer **dest**  
(destination) should be  
large enough to contain  
the concatenated  
resulting string

```
whovian@phy504:~/host$ gcc char14.c -o char14
whovian@phy504:~/host$ ./char14
Resulting string =
These strings are concatenated.
```

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

// char *strncat(char* dest,
//                const char* src,
//                size_t n)

// n = maximum num of chars
// to be appended.

// returns a pointer to the
// resulting string dest.

int main ()
{
    char str[80];
    strcpy (str, "these ");
    strncat (str, "strings ", 20);
    strncat (str, "are ", 20);
    strncat (str, "concatenated.", 5);

    printf("Resulting string = \n %s\n", str);

    return 0;
}
```

# Strings in C

How to concatenate two strings?

Two options: **strcat** or **strncat**

Pointer **dest** (destination) should be large enough to contain the concatenated resulting string

```
whovian@phy504:~/host$ gcc char15.c -o char15
whovian@phy504:~/host$ ./char15
Resulting string =
these strings are concat
```

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

// int strcmp(const char *leftStr,
//            const char *rightStr);

// code source: geeksforgeeks
// return < 0, 0, > 0
// > 0 -> left > right
// < 0 -> left < right

int main()
{
    char* leftStr = "g f g";
    char* rightStr = "g f g";

    const int res =
        strcmp(leftStr, rightStr);

    if (res==0)
    {
        printf("Strings are equal\n");
    }
    else
    {
        printf("Strings are unequal\n");
    }

    printf("Value returned by strcmp() is: %d\n", res);

    return 0;
}
```

# Strings in C

## How to compare two strings (dictionary)?

Two ways: **strcmp** (locale independent) and **strcoll** (advanced as it considers language locale)

```
whovian@phy504:~/host$ gcc char16.c -o char16
whovian@phy504:~/host$ ./char16
Strings are equal
Value returned by strcmp() is: 0
```

# Strings in C

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

int main()
{
    setlocale(LC_ALL, "en_US.UTF-8");

    char* leftStr = "é";
    char* rightStr = "f";

    const int res = strcmp(leftStr, rightStr);
    const int res2 = strcoll(leftStr, rightStr);

    printf(
        "Value returned by strcmp() is: %d\n" , res);

    printf(
        "Value returned by strcoll() is: %d\n" , res2);

    return 0;
}
```

Need to set the  
locale

How to compare two  
strings (dictionary)?

Two ways: **strcmp** (locale  
independent) and **strcoll**  
(advanced as it considers  
language locale)

```
whovian@phy504:~/host$ gcc char17.c -o char17
whovian@phy504:~/host$ ./char17
Value returned by strcmp() is: 93 ←
Value returned by strcoll() is: -68 ←
```

```

#include <stdio.h>
#include <string.h>

// memmove copies bytes (it does not
// care if the bytes are strings or not)
// void* memmove (void* destination,
//                 const void* source,
//                 size_t num );

// source: geeksforgeeks
int main()
{
    char str1[100] = "Geeks";
    char str2[10] = "Quiz";

    printf("str1 before memmove %s \n", str1);

    // make sure that memory has enough mem
    printf("size of str1: %lu \n", sizeof(str1));
    printf("size of str2: %lu \n", sizeof(str2));

    memmove(str1, str2, sizeof(str2));

    printf("str1 after memmove %s \n", str1);

    return 0;
}

```

# Strings in C (Copying)

```

whovian@phy504:~/host$ gcc char19.c -o char19
whovian@phy504:~/host$ ./char19
str1 before memmove Geeks
size of str1: 100
size of str2: 10
str1 after memmove Quiz

```

There are important differences between **memmove** and **memcpy**  
 (imp difference when strings overlap)

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

// memmove copies bytes (it does not
// care if the bytes are strings or not)
// void* memmove (void* destination,
//                 const void* source,
//                 size_t num );

// source: geeksforgeeks
int main()
{
    char str1[100] = "Geeks";
    char str2[10] = "Quiz";

    printf("str1 before memmove %s \n", str1);

    // make sure that memory has enough mem
    printf("size of str1: %lu \n", sizeof(str1));
    printf("size of str2: %lu \n", sizeof(str2));

    memmove(str1, str2, sizeof(str2));

    printf("str1 after memmove %s \n", str1);

    return 0;
}
```

# Strings in C (Copying)

```
whovian@phy504:~/host$ gcc char19.c -o char19
whovian@phy504:~/host$ ./char19
str1 before memmove Geeks
size of str1: 100
size of str2: 10
str1 after memmove Quiz
```

Why **printf** only prints  
**“Quiz”?**

```

#include <stdio.h>
#include <string.h>

// memmove copies bytes (it does not
// care if the bytes are strings or not)
// void* memmove (void* destination,
//                 const void* source,
//                 size_t num );

// source: geeksforgeeks
int main()
{
    char str1[100] = "GeeksforGeeks";
    char str2[10] = "Quiz";

    printf("str1 before memmove %s \n", str1);

    // make sure that memory has enough mem
    printf("size of str1: %lu \n", sizeof(str1));
    printf("size of str2: %lu \n", sizeof(str2));

    printf("str1 before memmove %s \n", str1);
    memmove(str1, str2, sizeof(str2));

    printf("str1 after memmove %s \n", str1);

    return 0;
}

```

# Strings in C (Copying)

```

whovian@phy504:~/host$ gcc char19.c -o char19
whovian@phy504:~/host$ ./char19
str1 before memmove GeeksforGeeks
size of str1: 100
size of str2: 10
str1 before memmove GeeksforGeeks
str1 after memmove Quiz

```

Why **printf** prints  
“**Quiz**”?

Strings always ends in the null char. **%s** only prints until the null char

String != char array

```

#include <stdio.h>
#include <string.h>

// memmove copies bytes (it does not
// care if the bytes are strings or not)
// void* memmove (void* destination,
//                 const void* source,
//                 size_t num );

// source: geeksforgeeks
int main()
{
    char str1[100] = "GeeksforGeeks";
    char str2[10] = "Quiz";

    printf("str1 before memmove %s \n", str1);

    // make sure that memory has enough mem
    printf("size of str1: %lu \n", sizeof(str1));
    printf("size of str2: %lu \n", sizeof(str2));

    memmove(str1, str2, sizeof(str2));

    printf("str1 after memmove %s \n", str1);

    printf("str1 with fwrite ");
    fwrite(str1, sizeof(char), 20, stdout);
    printf("\n");

    return 0;
}

```

# Strings in C (Copying)

```

whovian@phy504:~/host$ gcc char19v2.c -o char19v2
whovian@phy504:~/host$ ./char19v2
str1 before memmove GeeksforGeeks
size of str1: 100
size of str2: 10
str1 after memmove Quiz
str1 with fwrite Quizeks

```

Why **printf** prints “**Quiz**

Strings always end in the null char. **%s** only prints until the null char

String != char array

**fwrite** forces printing of the char array

```
#include "stdio.h"

int main(void)
{
    int length = 5;

    char string[length];
    string[0] = 'A';
    string[1] = 'B';
    string[2] = '\0'; // null char
    string[3] = 'C';
    string[4] = 'D';

    printf("With printf: %.*s\n", length,
           string);

    printf("With fwrite: ");
    fwrite(string, sizeof(char), length,
           stdout);
    printf("\n");

    return 0;
}
```

# Strings in C (Copying)

```
whovian@phy504:~/host$ gcc char19v3.c -o char19v3
whovian@phy504:~/host$ ./char19v3
With printf: AB
With fwrite: ABCD
```

Strings always end in the null char. **%s** only prints until the null char

String != char array

**fwrite** forces printing of the char array

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

int main()
{
    char str[100] = "Learningisfun";

    printf("original str: %s \n", str);

    // undefined behavior to use memcpy
    // memmove has an internal buffer, so
    // it is ok!
    memmove(str + 8, str, 15);
    printf("memmove overlap : %s \n", str);

    return 0;
}
```

```
whovian@phy504:~/host$ gcc char20.c -o char20
whovian@phy504:~/host$ ./char20
original str: Learningisfun
memmove overlap : LearningLearningisfun
```

# Strings in C (Copying)

There are important differences between  
**memmove and memcpy**  
(ex: when strings overlap)



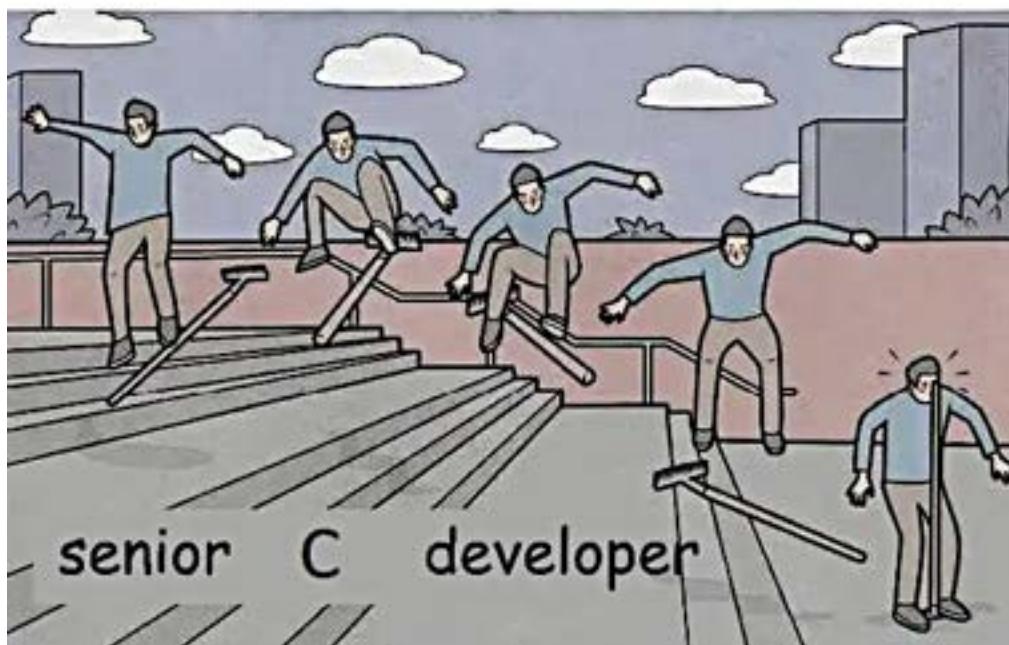
# Lecture 20: C part XI



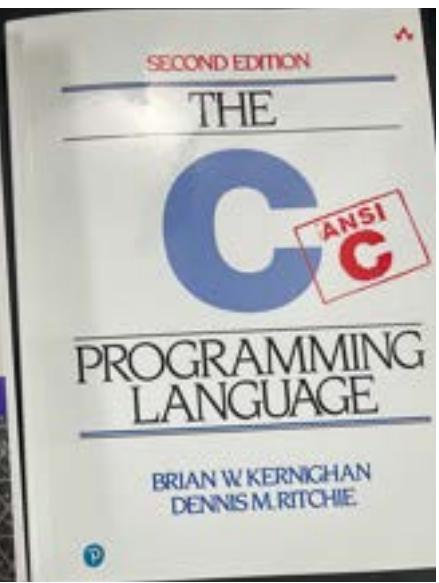
junior C developer



## Suggested Literature



senior C developer



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

// FILE *fopen(const char *file_name,
//              const char *mode_of_operation);
// -----
// modes of operation:
// - read only ("r")
// - write only ("w").
//     Original contents are overwritten
// - appending ("a")
// - read and write ("r+")
// - write but also read the content created
//     by writing ("w+"). Original contents
//     are overwritten
// - reading + appending ("a+")
// - read in binary mode ("rb")
// - writ in binary mode ("wb")

int main()
{
    // Creates a file with access as write-plus mode
    FILE* demo = fopen("demo_file.txt", "w+");

    fprintf(demo, "%s v%.2f", "Welcome to PHY504", 2.0);

    fclose(demo); // closes the file

    return 0;
}
```

# Working with files in C

C has specific libraries to open files directly (instead of relying on bash input / output redirection)

```
whovian@phy504:~/host$ gcc file.c -o file
whovian@phy504:~/host$ ./file
whovian@phy504:~/host$ more demo_file.txt
Welcome to PHY504 v2.00
```

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

int main()
{
    char str[20];
    strcpy(str, "Hello, Phy504!\n");

    FILE* fp = fopen("file4test.txt", "w");
    fputs(str, fp);
    fputs(str, stdout); // terminal
    fclose(fp);

    return 0;
}
```

# Working with files in C

fputs is an alternative to fprintf

## fputs vs fprintf

```
whovian@phy504:~/host$ gcc file4.c -o file4
whovian@phy504:~/host$ ./file4
Hello, Phy504!
whovian@phy504:~/host$ more file4test.txt
Hello, Phy504!
```

# Working with files in C

```
#include <stdio.h>

int main(void)
{
    FILE* ptr = fopen("file5test.txt", "r");

    char names[50];
    char ages[50];
    char cities[50];

    // return value fscanf: number of fields that
    // it successfully converted and assigned
    while (fscanf(ptr, "%s %s %s", names,ages,cities) == 3)
    {
        printf("%s %s %s \n", names, ages, cities);
    }

    return 0;
}
```

```
whovian@phy504:~/host$ gcc file5.c -o file5
whovian@phy504:~/host$ more file5test.txt
NAME      AGE     BORN CITY
Vivian    37     Rio-de-Janeiro
Beyonce   41     Houston
Rihanna   35     Saint-Michael
Nicki-Minaj 40   Saint-James
whovian@phy504:~/host$ ./file5
NAME AGE BORN CITY
Vivian 37 Rio-de-Janeiro
Beyonce 41 Houston
Rihanna 35 Saint-Michael
Nicki-Minaj 40 Saint-James
```

Like **scanf**, **fscanf** returns **int** = number of variables it read successfully. Use with **while** to do format reading of tables

# Working with files

```
#include <stdio.h>

// FILE* pointer to stream - can always be
// either the terminal or an actual file
// char *fgets(char *str, int n, FILE* stream)

int main(void)
{
    const int sz = 200;
    char buf[sz];

    // fgets always required a buffer
    // Always be careful with buffer overflow
    // (input string too large)
    // stdin = standard input (terminal)
    fgets(buf, sz, stdin);

    printf("string is: %s\n", buf);

    return 0;
}
```

**fgets** can be used  
with terminal or file

Alternative  
**char\* gets (char\* str)**  
assumes the  
terminal output

```
whovian@phy504:~/host$ gcc file3.c -o file3
whovian@phy504:~/host$ ./file3
Hello Phy504
string is: Hello Phy504
```

# Working with files

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

int main ()
{
    FILE* ptr = fopen ("unexist.ent", "r");

    if (ptr == NULL)
    {
        printf ("Error: %s \n", strerror(errno));
    }

    return 0;
}
```

How to convert  
**int** errors to  
human readable  
messages  
(in the std  
library)?

**errno, strerror**

```
whovian@phy504:~/host$ gcc file2.c -o file2
whovian@phy504:~/host$ ./file2
Error: No such file or directory
```

# Working with files

<b>errno value</b>	<b>Error</b>
1	/* Operation not permitted */
2	/* No such file or directory */
3	/* No such process */
4	/* Interrupted system call */
5	/* I/O error */
6	/* No such device or address */
7	/* Argument list too long */
8	/* Exec format error */
9	/* Bad file number */
10	/* No child processes */
11	/* Try again */
12	/* Out of memory */
13	/* Permission denied */

This way of handling error is **not safe** when doing parallelism

Why?  
global variable

**SOURCE:** <https://www.geeksforgeeks.org/error-handling-c-programs/>

# Working with files

**fseek** allows the user to position the file pointer in different places

```
GNU nano 4.8
#include <stdio.h>

// source: https://www.educba.com/fseek-in-c/
// int fseek(FILE *stream,
//           long int offset,
//           int pos)
// SEEK_CUR: current position of the ptr of the file.
// SEEK_END: moves the file ptr to the end of the file.
// SEEK_SET: moves the file ptr to the beginning of the file

int main()
{
    FILE* fx = fopen("file6test.txt", "r");

    fseek(fx, 0, SEEK_END);
    printf("Position of file pointer is : %ld \n", ftell(fx));

    fseek(fx, 10, SEEK_SET);

    int ch;
    printf("Resulting bytes after the 10 chars: ");
    while((ch = fgetc(fx)) != EOF) // fgetc get chars from file
    {
        putchar(ch);
    }
    return 0;
}
```

whovian@phy504:~/host\$ more file6test.txt  
Welcome to Phy504, everyone. Are you learning the C language?  
whovian@phy504:~/host\$ gcc file6.c -o file6  
whovian@phy504:~/host\$ ./file6  
Position of file pointer is : 62  
Resulting bytes after the 10 chars: Phy504, everyone. Are you learning the C language?

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

// source:
// www.fundza.com/c4serious/fileIO_reading_all/4_IO_readall.c

int main(void)
{
    FILE* infile = fopen("file7test.txt", "r");

    fseek(infile, 0L, SEEK_END); // 0L = 0 in long int
    long int numbytes = ftell(infile); ←
    fseek(infile, 0L, SEEK_SET); // reset the file pos

    // grab sufficient memory for the buffer
    char* buffer = (char*)calloc(numbytes, sizeof(char));

    // copy all the text into the buffer Char sz = 1 byte
    fread(buffer, sizeof(char), numbytes, infile);
    fclose(infile);

    // confirm we have read the file by outputting it
    printf("The file contains this text\n\n%s", buffer);

    free(buffer);
    return 0;
}
```

# Working with files

Reading entire file into memory at once (faster)

**ftell** returns long int (current position of the stream in bytes)

```
whovian@phy504:~/host$ more file7test.txt
This is the test file
Welcome to Phy504!
Here we will load the entire file into memory
whovian@phy504:~/host$ gcc file7.c -o file7
whovian@phy504:~/host$ ./file7
The file contains this text

This is the test file
Welcome to Phy504!
Here we will load the entire file into memory
```

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

// source:
// www.fundza.com/c4serious/fileIO_reading_all/4_IO_readall.c

int main(void)
{
    FILE* infile = fopen("file7test.txt", "r");

    fseek(infile, 0L, SEEK_END); // 0L = 0 in long int
    long int numbytes = ftell(infile);
    fseek(infile, 0L, SEEK_SET); // reset the file pos

    // grab sufficient memory for the buffer
    char* buffer = (char*)calloc(numbytes, sizeof(char));

    // copy all the text into the buffer 3 arg = total size
    fread(buffer, sizeof(char), numbytes, infile); ←
    fclose(infile); 2 arg = size of each element

    // confirm we have read the file by outputting it
    printf("The file contains this text\n\n%s", buffer);

    free(buffer);
    return 0;
}
```

# Working with files

Reading entire file into memory at once (faster)

**fread, fwrite** are lower-level functions

```
whovian@phy504:~/host$ more file7test.txt
This is the test file
Welcome to Phy504!
Here we will load the entire file into memory
whovian@phy504:~/host$ gcc file7.c -o file7
whovian@phy504:~/host$ ./file7
The file contains this text

This is the test file
Welcome to Phy504!
Here we will load the entire file into memory
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// source:
// https://www.geeksforgeeks.org/readwrite-structure-file-c/

struct person
{
    int id;
    char fname[20];
    char lname[20];
};

int main ()
{
    FILE* outfile = fopen ("file8test.txt", "w");

    struct person input1 = {1, "vivian", "Miranda"};
    struct person input2 = {2, "Wayne", "Hu"};

    // write struct to file
    // to visualize in bash: xxd file8test.txt
    fwrite (&input1, sizeof(struct person), 1, outfile);
    fwrite (&input2, sizeof(struct person), 1, outfile);

    fclose (outfile);      3 arg = total size
    return 0;              2 arg = size of each elem
}

```

# Working with files

**fwrite** is lower-level functions  
(deal with bytes).

Can be useful when writing **structs** as well

```

whovian@phy504:~/host$ gcc file8.c -o file8
whovian@phy504:~/host$ ./file8
whovian@phy504:~/host$ xxd file8test.txt
00000000: 0100 0000 7669 7669 616e 0000 0000 0000 ....vivian.....
00000010: 0000 0000 0000 0000 4d69 7261 6e64 6100 .....Miranda.
00000020: 0000 0000 0000 0000 0000 0000 0200 0000 .....
00000030: 5761 796e 6500 0000 0000 0000 0000 0000 Wayne.....
00000040: 0000 0000 4875 0000 0000 0000 0000 0000 ....Hu.....
00000050: 0000 0000 0000 0000 ..... 
```

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

// source:
// https://www.geeksforgeeks.org/readwrite-structure-file-c/

struct person
{
    int id;
    char fname[20];
    char lname[20];
};

int main ()
{
    // read mode
    FILE* infile = fopen("file8test.txt", "r");

    struct person input;  2 arg = size of each elem
                        3 arg = total size
    while(fread(&input, sizeof(struct person), 1, infile))
    {
        printf ("id = %d name = %s %s\n", input.id,
               input.fname, input.lname);
    }
    return 0;
}  return = num
                           elem read
```

# Working with files

**fwrite** is lower-level functions (deal with bytes).

Can be useful when writing **structs** as well

```
whovian@phy504:~/host$ gcc file9.c -o file9
whovian@phy504:~/host$ ./file9
file9  file9.c
whovian@phy504:~/host$ ./file9
id = 1 name = vivian Miranda
id = 2 name = Wayne Hu
```

```

#include <math.h>
#include <stdio.h>

int main()
{
    const double number = 4.0;

    const double squareRoot = sqrt(number);
    const double squared = pow(number, 2);
    const double cube = pow(number, 3);

    const double sinnum = sin(number); // number in rads
    const double cosnum = cos(number); // number in rads
    const double tannum = tan(number); // number in rads

    const double lognum = log(number); // natural log
    const double log10num = log10(number); //log_10

    printf("Sqrt of %.2lf = %.2lf \n", number, squareRoot);
    printf("Square of %.2lf = %.2lf \n", number, squared);
    printf("Cube of %.2lf = %.2lf \n", number, cube);

    printf("sin(%lf) = %.2lf \n", number, sinnum);
    printf("cos(%lf) = %.2lf \n", number, cosnum);
    printf("tan(%lf) = %.2lf \n", number, tannum);

    printf("log(%lf) = %.2lf \n", number, lognum);
    printf("log10(%lf) = %.2lf \n", number, log10num);

    return 0;
}

```

# Standard Library: math.h

Many useful math  
functions

Need manual linking  
with **-lm**



```

whovian@phy504:~/host$ gcc math.c -o math -lm
whovian@phy504:~/host$ ./math
Sqrt of 4.00 = 2.00
Square of 4.00 = 16.00
Cube of 4.00 = 64.00
sin(4.00) = -0.76
cos(4.00) = -0.65
tan(4.00) = 1.16
log(4.00) = 1.39
log10(4.00) = 0.60

```

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

int main (void)
{
    printf("The max value of float = %.5e\n", FLT_MAX);
    printf("The min value of float = %.5e\n\n", FLT_MIN);
    printf("The max value of double = %.5e\n", DBL_MAX);
    printf("The min value of double = %.5e\n\n", DBL_MIN);
    printf("The min eps of a float num = %.5e\n", FLT_EPSILON);
    printf("The min eps of a double num = %.5e\n\n", DBL_EPSILON);

    // (a == b is a bad idea) even the smallest rounding error
    // will cause two floating point numbers to not be equal,
    // operator== is at high risk for returning false

    const double x = 1.0;
    const double y = 1.0 + DBL_EPSILON/2.0;
    // don't try a == b
    if ((x/y - 1.0) < DBL_EPSILON) ←
    {
        printf("x and y are equal in practice\n");
    }
    else
    {
        printf("x and y are NOT equal\n");
    }
    return 0;
}
```

# Standard Library: **float.h**

## Limits on **float** / **double** types & proper float number comparison

```
whovian@phy504:~/host$ gcc float1.c -o float1
whovian@phy504:~/host$ ./float1
The max value of float = 3.40282e+38
The min value of float = 1.17549e-38

The max value of double = 1.79769e+308
The min value of double = 2.22507e-308

The min eps of a float num = 1.19209e-07
The min eps of a double num = 2.22045e-16

x and y are equal in practice
```

# Standard Library: **limits.h**

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

int main()
{
    printf("The min value of INT = %d\n", INT_MIN);
    printf("The max value of INT = %d\n", INT_MAX);

    printf("The min value of CHAR = %d\n", CHAR_MIN);
    printf("The max value of CHAR = %d\n", CHAR_MAX);

    printf("The min value of LONG = %ld\n", LONG_MIN);
    printf("The max value of LONG = %ld\n", LONG_MAX);

    return 0;
}
```

Important  
limits on other  
types such as  
**char**, **int** and  
**long int**

```
whovian@phy504:~/host$ nano limits1.c
whovian@phy504:~/host$ gcc limits1.c -o limits1
whovian@phy504:~/host$ ./limits1
The min value of INT = -2147483648
The max value of INT = 2147483647
The min value of CHAR = -128
The max value of CHAR = 127
The min value of LONG = -9223372036854775808
The max value of LONG = 9223372036854775807
```

```
#include <stdio.h>
#include <ctype.h>
// source: tutorialspoint.com/

int main (void)
{
    const int var1 = 'd';
    const int var2 = '2';
    const int var3 = '\t';
    const int var4 = ' ';

    if(isalnum(var1)) {
        printf("var1 = [%c] is alphanumeric\n", var1 );
    }
    else {
        printf("var1 = [%c] is not alphanumeric\n", var1 );
    }
    if(isalnum(var2)) {
        printf("var2 = [%c] is alphanumeric\n", var2 );
    }
    else {
        printf("var2 = [%c] is not alphanumeric\n", var2 );
    }
    if(isalnum(var3)) {
        printf("var3 = [%c] is alphanumeric\n", var3 );
    } else {
        printf("var3 = [%c] is not alphanumeric\n", var3 );
    }
    if(isalnum(var4)) {
        printf("var4 = [%c] is alphanumeric\n", var4 );
    } else {
        printf("var4 = [%c] is not alphanumeric\n", var4 );
    }
    return 0;
}
```

# Standard Library: **ctype.h**

```
whovian@phy504:~/host$ nano ctype1.c
whovian@phy504:~/host$ gcc ctype1.c -o ctype1
whovian@phy504:~/host$ ./ctype1
var1 = [d] is alphanumeric
var2 = [2] is alphanumeric
var3 = [ \t ] is not alphanumeric
var4 = [ ] is not alphanumeric
```

Check if a type is  
**alphanumeric** or  
**alphabetic** or **numeric** or  
**printable** (additional  
options and checks  
available on **ctype.h**)

Source: <https://www.tutorialspoint.com>

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

int main ()
{
    const int var1 = 'd';
    const int var2 = '2';
    const int var3 = '\t';
    const int var4 = ' ';

    if( isalpha(var1) ) {
        printf("var1 = %c is an alphabet\n", var1 );
    } else {
        printf("var1 = %c is not an alphabet\n", var1 );
    }

    if( isalpha(var2) ) {
        printf("var2 = %c is an alphabet\n", var2 );
    } else {
        printf("var2 = %c is not an alphabet\n", var2 );
    }

    if( isalpha(var3) ) {
        printf("var3 = %c is an alphabet\n", var3 );
    } else {
        printf("var3 = %c is not an alphabet\n", var3 );
    }

    if( isalpha(var4) ) {
        printf("var4 = %c is an alphabet\n", var4 );
    } else {
        printf("var4 = %c is not an alphabet\n", var4 );
    }

    return 0;
}
```

# Standard Library: **ctype.h**

```
whovian@phy504:~/host$ gcc ctype2.c -o ctype2
whovian@phy504:~/host$ ./ctype2
var1 = d is an alphabet
var2 = 2 is not an alphabet
var3 = \t is not an alphabet
var4 = ! is not an alphabet
```

Check if a type is  
**alphanumeric** or  
**alphabetic** or **numeric**  
or **printable** (additional  
options and checks  
available on **ctype.h**)

Source:

[https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_isalpha.htm](https://www.tutorialspoint.com/c_standard_library/c_function_isalpha.htm)

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

int main () {
    const int var1 = 'k';
    const int var2 = '8';
    const int var3 = '\t';
    const int var4 = ' ';

    if( isprint(var1) ) {
        printf("var1 = %c| can be printed\n", var1 );
    } else {
        printf("var1 = %c| can't be printed\n", var1 );
    }

    if( isprint(var2) ) {
        printf("var2 = %c| can be printed\n", var2 );
    } else {
        printf("var2 = %c| can't be printed\n", var2 );
    }

    if( isprint(var3) ) {
        printf("var3 = %c| can be printed\n", var3 );
    } else {
        printf("var3 = %c| can't be printed\n", var3 );
    }

    if( isprint(var4) ) {
        printf("var4 = %c| can be printed\n", var4 );
    } else {
        printf("var4 = %c| can't be printed\n", var4 );
    }

    return 0;
}
```

# Standard Library: **ctype.h**

```
whovian@phy504:~/host$ gcc ctype3.c -o ctype3
whovian@phy504:~/host$ ./ctype3
var1 = |k| can be printed
var2 = |8| can be printed
var3 = |          | can't be printed
var4 = | | can be printed
```

Check if a type is  
**alphanumeric** or  
**alphabetic** or **numeric**  
or **printable** (additional  
options and checks  
available on **ctype.h**)

Source:

[https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_isprint.htm](https://www.tutorialspoint.com/c_standard_library/c_function_isprint.htm)

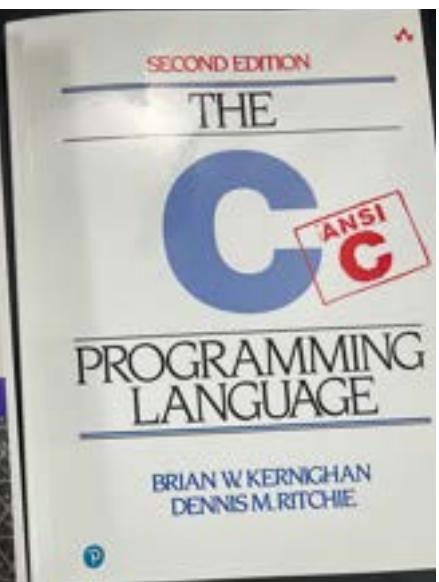
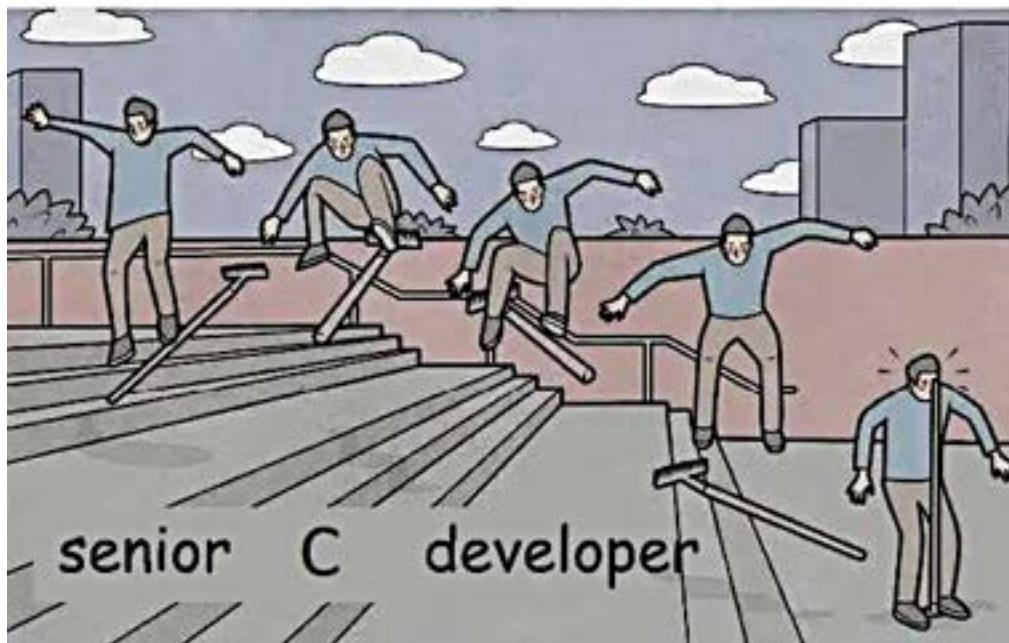
# Lecture 21: C part XII



junior C developer



## Suggested Literature



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

// source: geeksforgeeks. C program for finding the
// largest int among 3 ints using command line args

int main(int argc, char** argv)
{
    if (argc < 4 || argc > 5)
    { // the program itself is an argument
        printf("Please enter 3 integers only\n");
        printf("You entered %i integers\n", argc-1);
        return 1;
    }
    const int a = atoi(argv[1]);
    const int b = atoi(argv[2]);
    const int c = atoi(argv[3]);

    if (a < 0 || b < 0 || c < 0)
    {
        printf("Please enter only positive values\n");
        return 1;
    }
    if (!(a != b && b != c && a != c))
    {
        printf("Please enter 3 different values\n");
        return 1;
    }
    else
    {
        if (a > b && a > c)
            printf("%d is largest\n", a);
        else if (b > c && b > a)
            printf ("%d is largest\n", b);
        else if (c > a && c > b)
            printf(" %d is largest\n",c);
    }
    return 0;
}
```



# Arguments in the main function

- **int argc**: the number of arguments. The program itself count as an argument

```
whovian@phy504:~/host$ ./commandline1
Please enter 3 integers only
You entered 0 integers
whovian@phy504:~/host$ ./commandline1 1
Please enter 3 integers only
You entered 1 integers
whovian@phy504:~/host$ ./commandline1 1 2
Please enter 3 integers only
You entered 2 integers
```

Source: <https://www.geeksforgeeks.org/find-largest-among-three-different-positive-numbers-using-command-line-argument/>

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

// source: geeksforgeeks. C program for finding the
// largest int among 3 ints using command line args

int main(int argc, char** argv)
{
    if (argc < 4 || argc > 5)
    { // the program itself is an argument
        printf("Please enter 3 integers only\n");
        printf("You entered %i integers\n", argc-1);
        return 1;
    }
    const int a = atoi(argv[1]);
    const int b = atoi(argv[2]);
    const int c = atoi(argv[3]);

    if (a < 0 || b < 0 || c < 0)
    {
        printf("Please enter only positive values\n");
        return 1;
    }
    if (!(a != b && b != c && a != c))
    {
        printf("Please enter 3 different values\n");
        return 1;
    }
    else
    {
        if (a > b && a > c)
            printf("%d is largest\n", a);
        else if (b > c && b > a)
            printf ("%d is largest\n", b);
        else if (c > a && c > b)
            printf("%d is largest\n",c);
    }
    return 0;
}
```

# Arguments in the main function

- **char\*\* argv**: an array of strings. In this case we need to convert them to integers

```
whovian@phy504:~/host$ ./commandline1 1 2 3
3 is largest
whovian@phy504:~/host$ ./commandline1 1 3 3
Please enter 3 different values
whovian@phy504:~/host$ ./commandline1 1 3 -10
Please enter only positive values
whovian@phy504:~/host$ ./commandline1 50 3 2
50 is largest
```

Source: <https://www.geeksforgeeks.org/find-largest-among-three-different-positive-numbers-using-command-line-argument/>

```

#include <math.h>
#include <stdio.h>
#include <time.h>

// source:
// www.geeksforgeeks.org/time-h-header-file-in-c-with-examples/

int frequency_of_primes(int n)
{ // the number of primes less than the given param
    int freq = n - 1;
    for (int i = 2; i <= n; ++i)
    {
        for (int j = sqrt(i); j > 1; --j)
        {
            if (i % j == 0)
            {
                --freq;
                break;
            }
        }
    }
    return freq;
}
int main()
{
    clock_t t = clock();
    const int f = frequency_of_primes(99999);

    printf("The number of primes lower"
           " than 10,000 is: %d\n", f);

    t = clock() - t;

    printf("No. of clicks %ld clicks (%f seconds).\n",
           t, ((float) t) / CLOCKS_PER_SEC);
    return 0;
}

```

# Standard Library: time.h

Functions to display local (or GMT) time and to measure execution elapsed time

```

whovian@phy504:~/host$ gcc time1.c -o time1 -lm
whovian@phy504:~/host$ ./time1
The number of primes lower than 10,000 is: 9592
No. of clicks 92552 clicks (0.092552 seconds).

```

More examples  
(including display of local / UTC time) via geeksforgeeks

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

int main () {
    char str[20];

    strcpy(str, "98993489");
    const int val = atoi(str);
    printf("String value = %s, Int val = %d\n", str, val);

    strcpy(str, "tutorialspoint.com");
    const int val2 = atoi(str); // when failing, returns 0
    printf("String value = %s, Int val = %d\n", str, val2);

    strcpy(str, "2.3");
    const double val3 = atof(str);
    printf("String value = %s, Float val = %lf\n", str, val3);

    strcpy(str, "tutorialspoint.com");
    const double val4 = atof(str); // when failing, returns 0
    printf("String value = %s, Float val = %lf\n", str, val4);

    strcpy(str, "9899348912434");
    const long int val5 = atol(str);
    printf("String value = %s, Long int val = %ld\n", str, val5);

    strcpy(str, "tutorialspoint.com");
    const long int val6 = atol(str); // when failing, returns 0
    printf("String value = %s, Long int val = %ld\n", str, val6);

    return 0;
}

whovian@phy504:~/host$ ./stdlib1
String value = 98993489, Int val = 98993489
String value = tutorialspoint.com, Int val = 0
String value = 2.3, Float val = 2.300000
String value = tutorialspoint.com, Float val = 0.000000
String value = 9899348912434, Long int val = 9899348912434
String value = tutorialspoint.com, Float val = 0
```

# Standard Library: **stdlib.h**

Many important  
functions

- **malloc/calloc/realloc**
- **free**
- **exit(), abort()**
- **int rand()**
- **atof, atoi, atol**
- **char\* getenv(const char \*name)**

Source:

[https://www.tutorialspoint.com/c\\_standard\\_library/stdlib\\_h.htm](https://www.tutorialspoint.com/c_standard_library/stdlib_h.htm)

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

// double strtod(const char *str, char **endptr)
//
// str: string to be converted to a value
// endptr: reference to an already allocated object of type,
//          char*, whose value is set by the function to the
//          next char in str after the numerical value.

int main ()
{
    char str[30] = "20.30300 This is test";
    char *ptr;

    double num1 = strtod(str, &ptr);

    printf("The number(double) is %lf\n", num1);
    printf("String part is %s\n", ptr);

    double num2 = atof(str);
    printf("The number(double) via atof is %lf\n", num2);

    return(0);
}
```

# Standard Library:

## stdlib.h

Functions **atof**, **atoi**,  
**atoll** work ok but  
there are functions  
(e.g., **strtod**) that are  
more complete.

**strtod** provides mixed  
translation from string  
to number and words

```
whovian@phy504:~/host/C$ ./stdlib5
The number(double) is 20.303000
String part is | This is test|
The number(double) via atof is 20.303000
```

Source:

[https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_strtod.htm](https://www.tutorialspoint.com/c_standard_library/c_function_strtod.htm)

# Standard Library: **stdlib.h**

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

// double strtod(const char *str, char **endptr)
// this function (same for strtol) allows error
// checking (that is not the case of atof/atoi)

int main ()
{
    char str[30] = "This is test";
    char *ptr;

    double num1 = strtod(str, &ptr);
    if (ptr == str) // how to check errors
    {
        printf("Conversion of %s to double fail\n", str);
    }

    char str2[30] = "1234.0 This is test";
    char *ptr2;
    double num2 = strtod(str2, &ptr2);
    if (ptr2 != str2) // how to check errors
    {
        printf("Conversion of %s to double %lf succeed\n",
               str, num2);
    }
    return 0;
}
```

**strtod/strtol**  
functions also  
provide a way to  
check if the  
conversion was  
successful

```
whovian@phy504:~/host/C$ gcc stdlib7.c -o stdlib7
whovian@phy504:~/host/C$ ./stdlib7
Conversion of !This is test! to double fail
Conversion of !This is test! to double 1234.000000 succeed
```

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

// long int strtol(const char *str, char **endptr, int base)

// str: string to be converted to a value

// endptr: reference to an already allocated object of type,
//          char*, whose value is set by the function to the
//          next char in str after the numerical value.

// base: This is the base, which must be between 2 and 36
//        inclusive, or be the special value 0.

int main ()
{
    char str[30] = "-2030300 This is test";
    char *ptr;

    long int num1 = strtol(str, &ptr, 10);

    printf("The number (long int) is %ld\n", num1);
    printf("String part is |%s|\n", ptr);

    long int num2 = atoi(str);
    printf("The number (long int) via atoi is %ld\n", num2);

    return 0;
}
```

```
whovian@phy504:~/host/C$ ./stdlib6
The number (long int) is -2030300
String part is | This is test|
The number (long int) via atoi is -2030300
```

# Standard Library: stdlib.h

Functions **atof**, **atoi**, **atoll** work ok but there are functions (e.g., **strtol**) that are more complete

**strtod** provides mixed translation from string to number and words

Source:

[https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_strtol.htm](https://www.tutorialspoint.com/c_standard_library/c_function_strtol.htm)

# Standard Library: **stdlib.h**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// source:
// https://www.tutorialspoint.com/
// c_standard_library/c_function_rand.htm

int main (void)
{
    time_t t;
    // Initializes random number generator
    srand((unsigned) time(&t));

    for(int i=0; i<5 ; i++ )
    { // Print 5 random number [0,49]
        printf("%d\n", rand() % 50);
    }

    return 0;
}
```

**stdlib.h** provides a high-quality random number generator

- **srand** = provide the seed for the generator

```
whovian@phy504:~/host$ gcc stdlib2.c -o stdlib2
whovian@phy504:~/host$ ./stdlib2
27
26
27
19
44
whovian@phy504:~/host$ ./stdlib2
46
8
29
48
30
```

Source: [https://www.tutorialspoint.com/c\\_standard\\_library/stdlib\\_h.htm](https://www.tutorialspoint.com/c_standard_library/stdlib_h.htm)

# Standard Library: **stdlib.h**

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

int main ()
{
    printf("PATH : %s\n", getenv("PATH"));
    printf("HOME : %s\n", getenv("HOME"));
    printf("ROOT : %s\n", getenv("ROOT"));

    return 0;
}
```

**stdlib.h** provides the **getenv** function that allows **shell/bash** environmental variables to be read

```
whovian@phy504:~/host/C$ gcc stdlib3.c -o stdlib3
whovian@phy504:~/host/C$ ./stdlib3
PATH : /opt/conda/bin:/usr/local/nvidia/bin:/usr/local/cuda/bin:/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin
HOME : /home/whovian
ROOT : (null)
```

Source: [https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_getenv.htm](https://www.tutorialspoint.com/c_standard_library/c_function_getenv.htm)

```

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

// void qsort(void *base, size_t nitems, size_t size,
// int (*compar)(const void *, const void*))

int values[] = { 88, 56, 100, 2, 25 };

int cmpfunc (const void* a, const void* b)
{
    return ( *(int*)a - *(int*)b );
}

int main ()
{
    printf("Before sorting the list is:\n");
    for(int n=0; n<5; n++)
    {
        printf("%d ", values[n]);
    }
    printf("\n");

    qsort(values, 5, sizeof(int), cmpfunc);

    printf("After sorting the list is:\n");
    for (int n=0; n<5; n++)
    {
        printf("%d ", values[n]);
    }
    printf("\n");
    return 0;
}

```

# Standard Library:

## stdlib.h

**stdlib.h** provide generic functions such as **qsort**.

Generic functions use void pointers because anything can be casted into **void\***

```

whovian@phy504:~/host/C$ gcc stdlib4.c -o stdlib4
whovian@phy504:~/host/C$ ./stdlib4
Before sorting the list is:
88 56 100 2 25
After sorting the list is:
2 25 56 88 100

```

Source:

[https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_qsort.htm](https://www.tutorialspoint.com/c_standard_library/c_function_qsort.htm)

# Headers in C

```
#include <stdio.h>

void print_int(const int x)
{
    printf("The number is: %i\n", x);
}

#ifndef __PHY504_TEST_HEADER_H
#define __PHY504_TEST_HEADER_H
#ifndef __cplusplus
extern "C" {
#endif

void print_int(const int x);

// declaration does not care about
// variable names (only care about type).
// So you can declare the function as
// void print_int(const int);

#ifndef __cplusplus
}
#endif
#endif // HEADER GUARD
```

File 1: **testheader1.c**

body of the function

header 1: **testheader1.h**

Include declarations of  
functions (and global  
variables / structs)

Headers need pre-  
processor guards

# Headers in C

```
#include <stdio.h>

void print_int(const int x)
{
    printf("The number is: %i\n", x);
}

#ifndef __PHY504_TEST_HEADER_H
#define __PHY504_TEST_HEADER_H
#endif __cplusplus
extern "C" {

void print_int(const int x);

// declaration does not care about
// variable names (only care about type).
// So you can declare the function as
// void print_int(const int);

#endif __cplusplus
#endif // HEADER GUARD
```

File 1: **testheader1.c**

body of the function

## Advanced technical point

If you mix C and C++ code, the C headers need these additional guards.

Why? Differences between C and C++ on something called “Name Mangling”

# Headers in C

```
#include "testheader1.h"

int main(void)
{
    const int myint = 10;
    print_int(myint);
    return 0;
}
```

File 2: **testheader1\_main.c**

Location of the main  
function

Why the double quotes in “**testheader1.h**”?

Trick that tells the pre-processor to look for the file in the  
current directory first

How do I compile the code? No need to include the header

```
whovian@phy504:~/host$ gcc testheader1_main.c testheader1.c -o testheader
whovian@phy504:~/host$ ./testheader
The number is: 10
```

# Static variables outside functions

```
#include <stdio.h>

#ifndef __PHY504_TEST_HEADER3_H
#define __PHY504_TEST_HEADER3_H
#ifndef __cplusplus
extern "C" {
#endif

// When you #include "testheader3.h" in each of a dozen files,
// the compiler will produce a new instance of the max function
// in each of them. But because you've given the function internal
// linkage, none of the files has made public the function name max,
// so all dozen separate instances of the function can live
// independently with no conflicts.

static inline void print_int(const int x)
{
    printf("The number is: %i\n", x);
}

#ifndef __cplusplus
}
#endif
#endif // HEADER GUARD
```

- Internal linkage = file's instance of a var x or a function f() is its own and matches only other instances of x or f() in the same scope. Use the static keyword to indicate internal linkage.

Source:

<https://stackoverflow.com/questions/7762731/whats-the-difference-between-static-and-static-inline-function>

Source2:

<https://stackoverflow.com/a/25000931>

# Static variables outside functions

```
27 static int GSL_WORKSPACE_SIZE = 250;
28 static int use_linear_ps_limber = 0; // 0 or 1
29
30 static int include_HOD_GX = 0; // 0 or 1
31
32 static int include_RSD_GS = 0; // 0 or 1
33 static int include_RSD_GG = 1; // 0 or 1
34 static int include_RSD_GK = 0; // 0 or 1
35 static int include_RSD_GY = 0; // 0 or 1
36
```

Example of static global variables with internal linkage used in my research. They are great for setting compile-time options that are relevant only in one file

Source: [https://github.com/CosmoLike/cocoa/blob/main/Cocoa/external\\_modules/code/cosmolike/cosmo2D.c](https://github.com/CosmoLike/cocoa/blob/main/Cocoa/external_modules/code/cosmolike/cosmo2D.c)

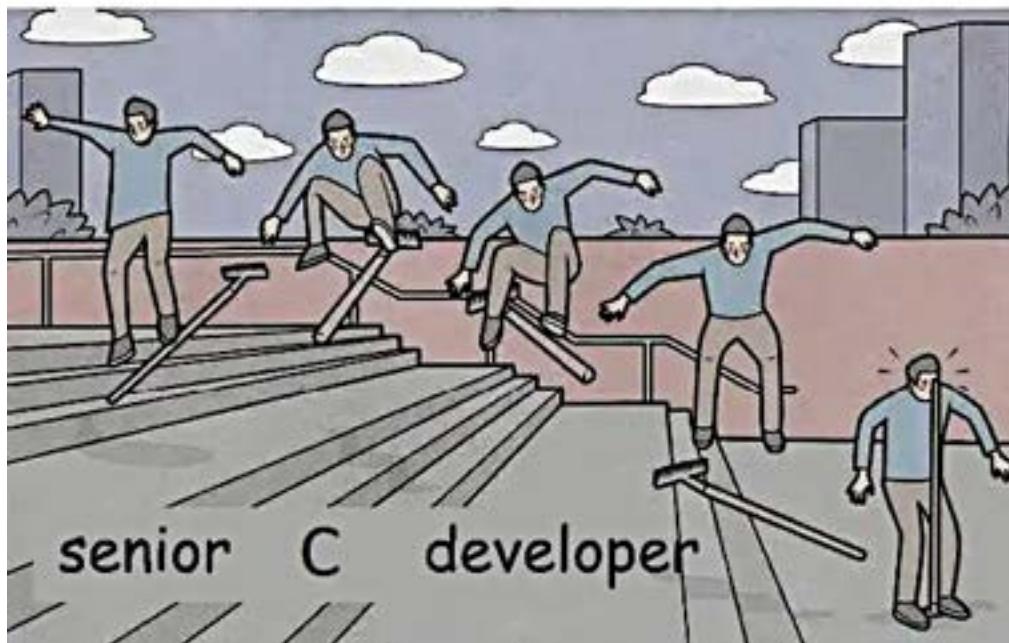
# Lecture 22: C part XIII



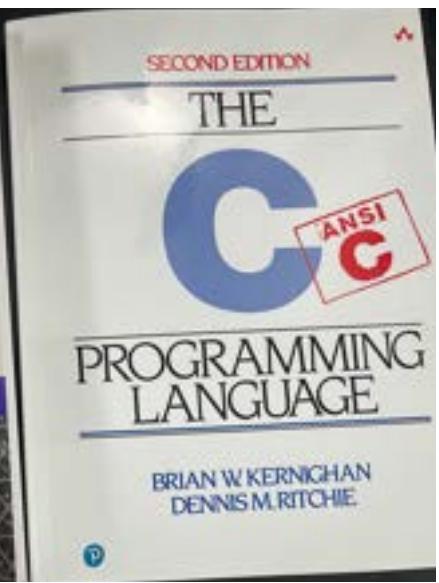
junior C developer



Suggested Literature



senior C developer



# Static variables outside functions

```
27 static int GSL_WORKSPACE_SIZE = 250;
28 static int use_linear_ps_limber = 0; // 0 or 1
29
30 static int include_HOD_GX = 0; // 0 or 1
31
32 static int include_RSD_GS = 0; // 0 or 1
33 static int include_RSD_GG = 1; // 0 or 1
34 static int include_RSD_GK = 0; // 0 or 1
35 static int include_RSD_GY = 0; // 0 or 1
36
```

Example of static global variables with internal linkage used in my research. They are great for setting compile-time options that are relevant only in one file

Source: [https://github.com/CosmoLike/cocoa/blob/main/Cocoa/external\\_modules/code/cosmolike/cosmo2D.c](https://github.com/CosmoLike/cocoa/blob/main/Cocoa/external_modules/code/cosmolike/cosmo2D.c)

# Static variables outside functions

```
GNU nano 6.2 teststatic2.c *
#include <stdio.h>

static int myglobal = 10;

void printglobal()
{
    printf("myglobal: %i\n", myglobal);
    myglobal += 50;
}
```

```
#include <stdio.h>
#include "teststatic2.h"
#include "teststatic3.h"

int main ()
{
    printglobal();
    printglobal2();

    printglobal();
    printglobal2();
}
```

```
GNU nano 6.2 teststatic3.c
#include <stdio.h>

static int myglobal = 20;

void printglobal2()
{
    printf("myglobal: %i\n", myglobal);
    myglobal++;
}
```

```
whovian@phy504:~/host/C$ gcc teststatic.c teststatic2.c teststatic3.c -o teststatic
whovian@phy504:~/host/C$ ./teststatic
myglobal: 10
myglobal: 20
myglobal: 60
myglobal: 21
```

They have the same name, but act independent because of the **static** prefix

# Static variables outside functions

```
GNU nano 6.2  teststatic2.c *
#include <stdio.h>

int myglobal = 10;

void printglobal()
{
    printf("myglobal: %i\n", myglobal);
    myglobal += 50;
}
```

```
GNU nano 6.2  teststatic3.c *
#include <stdio.h>

int myglobal = 20;

void printglobal2()
{
    printf("myglobal: %i\n", myglobal);
    myglobal++;
}
```

```
#include <stdio.h>
#include "teststatic2.h"
#include "teststatic3.h"

int main ()
{
    printglobal();
    printglobal2();

    printglobal();
    printglobal2();

}
```

```
whovian@phy504:~/host/C$ gcc teststatic.c teststatic2.c teststatic3.c -o teststatic
/usr/bin/ld: /tmp/ccs3VJzB.o:(.data+0x0): multiple definition of `myglobal'; /tmp/ccnekUmq.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Without **static** prefix, linking of the files done by the compiler fail

# How to make a global variable to all files?

```
GNU nano 6.2 teststatic4.h *
#ifndef PHY504_TESTSTATIC4_H
#define PHY504_TESTSTATIC4_H

void printglobal();

extern int myglobal;

#endif
```

```
GNU nano 6.2 teststatic5.h *
#ifndef PHY504_TESTSTATIC5_H
#define PHY504_TESTSTATIC5_H

void printglobal2();

#endif
```

```
GNU nano 6.2
#include <stdio.h>
#include "teststatic4.h"
#include "teststatic5.h"

int main ()
{
    printglobal();
    printglobal2();

    printglobal();
    printglobal2();
}
```

```
GNU nano 6.2 teststatic4.c
#include <stdio.h>

int myglobal = 10;

void printglobal()
{
    printf("myglobal: %i\n", myglobal);
    myglobal += 50;
}
```

```
GNU nano 6.2      teststatic5.c *
#include <stdio.h>
#include "teststatic4.h"

void printglobal2()
{
    printf("myglobal: %i\n", myglobal);
    myglobal++;
}
```

```
whovian@phy504:~/host/C$ gcc teststatic4.c teststatic5.c teststatic6.c -o teststatic6
whovian@phy504:~/host/C$ ./teststatic6
myglobal: 10
myglobal: 60
myglobal: 61
myglobal: 111
```

# Inline functions - Headers in C

- inline instructs the compiler to **attempt to** embed the function content into the calling code
- For small functions that are called frequently that can make a big performance difference.
- This is only a "hint", and the compiler may ignore it.  
From header also require **extern** for linkage

Source:

<https://stackoverflow.com/questions/7762731/whats-the-difference-between-static-and-static-inline-function>

Source2: <https://stackoverflow.com/a/25000931>

```
#include <stdio.h>

#ifndef __PHY504_TEST_HEADER2_H
#define __PHY504_TEST_HEADER2_H
#ifndef __cplusplus
extern "C" {
#endif

extern inline void print_int(const int x)
{
    printf("The number is: %i\n", x);
}

#endif __cplusplus
#endif
#endif // HEADER GUARD
```

# Inline functions - Headers in C

```
#include <stdio.h>

#ifndef __PHY504_TEST_HEADER2_H
#define __PHY504_TEST_HEADER2_H
#ifndef __cplusplus
extern "C" {
#endif

extern inline void print_int(const int x)
{
    printf("The number is: %i\n", x);
}

#ifndef __cplusplus
}
#endif
#endif // HEADER GUARD
```

This is only a "hint", and the compiler may ignore it. C also requires **extern** for linkage. Why?

External linkage means that symbols that match across files should be treated as the same thing by the linker. The **extern** keyword will be useful to indicate external linkage

Source:

<https://stackoverflow.com/questions/7762731/whats-the-difference-between-static-and-static-inline-function>

Source2: <https://stackoverflow.com/a/25000931>

# Static inline functions

```
#include <stdio.h>

#ifndef __PHY504_TEST_HEADER3_H
#define __PHY504_TEST_HEADER3_H
#ifndef __cplusplus
extern "C" {
#endif

// When you #include "testheader3.h" in each of a dozen files,
// the compiler will produce a new instance of the max function
// in each of them. But because you've given the function internal
// linkage, none of the files has made public the function name max,
// so all dozen separate instances of the function can live
// independently with no conflicts.

static inline void print_int(const int x)
{
    printf("The number is: %i\n", x);
}

#ifndef __cplusplus
#endif
#endif // HEADER GUARD
```

- Internal linkage = file's instance of a var **x** or a function **f()** is its own and matches only other instances of **x** or **f()** in the **same scope (file)**.

Use the static keyword to indicate internal linkage.

Source:

<https://stackoverflow.com/questions/7762731/what-is-the-difference-between-static-and-static-inline-function>

Source2: <https://stackoverflow.com/a/25000931>

# Static inline functions

```
GNU nano 6.2      testheader4.h
#include <stdio.h>

#ifndef __PHY504_TEST_HEADER4_H
#define __PHY504_TEST_HEADER4_H

static inline void print_int(const int x)
{
    printf("Print1: The number is: %i\n", x);
}

#endif // HEADER GUARD
```

```
GNU nano 6.2      testheader5.h
#ifndef __PHY504_TEST_HEADER5_H
#define __PHY504_TEST_HEADER5_H

void test_static_inline1();

#endif // HEADER GUARD
```

```
GNU nano 6.2      testheader5.c
#include "testheader4.h"

void test_static_inline1()
{
    print_int(10);
}
```

```
GNU nano 6.2      testheader6.h
#include <stdio.h>

#ifndef __PHY504_TEST_HEADER6_H
#define __PHY504_TEST_HEADER6_H

static inline void print_int(const int x)
{
    printf("Print2: The number is: %i\n", x);
}

#endif // HEADER GUARD
```

```
GNU nano 6.2      testheader6.c
#include "testheader5.h"
#include "testheader6.h"

int main()
{
    print_int(100);
    test_static_inline1();
}
```

```
whovian@phy504:~/host/C$ gcc testheader5.c testheader6.c -o testheader6
whovian@phy504:~/host/C$ ./testheader6
Print2: The number is: 100
Print1: The number is: 10
```

# Static inline functions

Without the static keyword, compile can't link the files

```
whovian@phy504:~/host/C$ gcc testheader8.c testheader9.c -o testheader6
/usr/bin/ld: /tmp/cc0Tkj0j.o: in function `test_static_inline1':
testheader8.c:(.text+0xe): undefined reference to `print_int'
/usr/bin/ld: /tmp/ccW2WQF7.o: in function `main':
testheader9.c:(.text+0xe): undefined reference to `print_int'
collect2: error: ld returned 1 exit status
```

How to create a single inline function? Keyword **extern**

```
GNU nano 6.2          testheader10.h
#include <stdio.h>

#ifndef __PHY504_TEST_HEADER10_H
#define __PHY504_TEST_HEADER10_H

extern inline void print_int(const int x)
{
    printf("The number is: %i\n", x);
}

#endif // HEADER GUARD
```

```
GNU nano 6.2          testheader10.c
#include "testheader10.h"

int main()
{
    print_int(100);
}
```

```
whovian@phy504:~/host/C$ gcc testheader10.c -o testheader10
whovian@phy504:~/host/C$ ./testheader10
The number is: 100
```

# Casting between const and non-const

```
#include <stdio.h>

// casting between const and not const
// not a good design but possible

// source: 21st Century C, 2nd Edition.
// Author: Ben Klemens
void set_elmt(int* a, int* b)
{
    a[0] = 3;
}

int main()
{
    int a[10] = {0,1,2,3,4,5,6,7,8,9};
    const int b[10] = {0,1,2,3,4,5,6,7,8,9};
    const int* c = b; // variable ptr to a const int

    set_elmt(a, (int*) c);

    printf("a[0] = %i, a[1] = %i\n", a[0], a[1]);
    printf("b[0] = %i, b[1] = %i\n", b[0], b[1]);
    return 0;
}
```

When working with functions (especially external functions), the const can cause API incompatibilities.

Casting is possible if the function don't break the const promise (otherwise you get undefined behavior)

```
whovian@phy504:~/host/C$ gcc castingconst.c -o castingconst
whovian@phy504:~/host/C$ ./castingconst
a[0] = 3, a[1] = 1
b[0] = 0, b[1] = 1
```

# Casting between const and non-const

```
#include <stdio.h>

// casting between const and not const
// not a good design but possible

// source: 21st Century C, 2nd Edition.
// Author: Ben Klemens
void set_elmt(int* a, int* b)
{
    a[0] = 3;
}

int main()
{
    int a[10] = {0,1,2,3,4,5,6,7,8,9};
    const int b[10] = {0,1,2,3,4,5,6,7,8,9};
    const int* c = b; // variable ptr to a const int

    set_elmt((int*) c, a); // casting from non-const to
                           // const is always ok

    printf("a[0] = %i, a[1] = %i\n", a[0], a[1]);
    printf("b[0] = %i, b[1] = %i\n", b[0], b[1]);
    return 0;
}
```

```
whovian@phy504:~/host/C$ ./castingconst2
a[0] = 0, a[1] = 1
b[0] = 3, b[1] = 1
whovian@phy504:~/host/C$ █
```

Seems to work fine with our gcc version but it is undefined behavior. Why?

Sometimes const variables sits in unwritable memory location (memory with read-only attributes)

Source:

<https://stackoverflow.com/a/1704922/2472169>



# GSL Gnu Scientific Library

Excellent free library for scientific computing



**GNU** Operating System

Supported by the Free Software Foundation

[PHILOSOPHY](#)   [LICENSES](#)   [EDUCATION](#)   [SOFTWARE](#)   [DISTROS](#)   [DOCS](#)   [MALWARE](#)   [HELP GNU](#)

## GSL - GNU Scientific Library

---

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License.

The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.

Unlike the licenses of proprietary numerical libraries the license of GSL does not restrict scientific cooperation. It allows you to share your programs freely with others.

Current stable version: [GSL-2.7](#)

```
#include <stdio.h>
#include <gsl/gsl_errno.h> // error handling
#include <gsl/gsl_sf_bessel.h>

int main (void)
{
    // Simplified Way to call GSL SF. Bessel Function J_0
    const double x = 5.0;
    const double y = gsl_sf_bessel_J0 (x);
    printf ("J0(%g) = %.18e\n", x, y);

    // Advanced way to call SF. The return int indicates
    // status of the call (only needed if not using the
    // default error handler that crashes the program)
    // int gsl_sf_bessel_J0_e(double x, gsl_sf_result* res)
    // gsl_sf_result contains the result and the error

    // By default, GSL crashes the program. With error
    // handler off, we need to check the the return status
    gsl_set_error_handler_off();

    gsl_sf_result res;
    int status = gsl_sf_bessel_J0_e(x, &res);
    if (status)
    {
        const char* emsg = gsl_strerror(status);
        printf("Error GSL J0 SF: %s. Location: %s, line %d\n",
               emsg, __FILE__, __LINE__);
        exit(1);
    }

    printf ("J0(%g) = %.18e. Error = %.18e\n",
           x, res.val, res.err);

    return 0;
}
```

# GSL

## Special Functions

GSL can compute many  
special functions

There are two interfaces: basic  
and advanced

Advanced interface provides  
two extra capabilities: manual  
error handling via **status**  
**integer** and evaluation of  
expected error

```
whovian@phy504:~/host/C$ gcc -O2 gsl1.c -o gsl1 -lgsl -lgslcblas -lm
whovian@phy504:~/host/C$ ./gsl1
J0(5) = -1.775967713143382642e-01
J0(5) = -1.775967713143382642e-01. Error = 1.930210957968419627e-16
```

# GSL

## Integration: adaptive

How to characterize the integrand (one variable)?

With **GSL FUNCTION** struct

```
struct gsl_function_struct
{
    double (* function) (double x, void * params);
    void * params;
};

typedef struct gsl_function_struct gsl_function ;
```

The **void\* params** allows you to send constants to the integrand, for example,  $f(x) = ax + b$  with  $a, b$  constants

# GSL

## Integration: adaptive

### Functions have addresses

Therefore, there is a  
function pointer

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void* params)
{
    double alpha = *(double *) params;
    return log(alpha*x) / sqrt(x);
}

int main (void)
{
    const size_t max_num_subdiv = 1000;
    // adaptive integration need memory allocation of a
    // special type of struct to hold intermediate values
    gsl_integration_workspace* w
        = gsl_integration_workspace_alloc (max_num_subdiv);

    double result, error, alpha = 1.0;
    gsl_function F;
    F.function = &f;
    F.params = &alpha; // automatic cast to void*

    const double int_min = 0.0, int_max = 1.0;
    const double rel_error = 1e-7, abs_error = 0.0;

    gsl_integration_qags (&F, int_min, int_max, abs_error,
        rel_error, max_num_subdiv, w, &result, &error);

    printf ("result      = %.18f\n", result);
    printf ("exact result = %.18f\n", -4.0);
    printf ("estimated error = %.18f\n", error);
    printf ("actual error   = %.18f\n", result - (-4.0));

    gsl_integration_workspace_free (w);
    return 0;
}
```

```
struct gsl_function_struct
{
    double (* function) (double x, void * params);
    void * params;
};

typedef struct gsl_function_struct gsl_function;
```

Each function type has its  
unique function pointer

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void* params)
{
    double alpha = *(double *) params;
    return log(alpha*x) / sqrt(x);
}

int main (void)
{
    const size_t max_num_subdiv = 1000;
    // adaptive integration need memory allocation of a
    // special type of struct to hold intermediate values
    gsl_integration_workspace* w
        = gsl_integration_workspace_alloc (max_num_subdiv);

    double result, error, alpha = 1.0;
    gsl_function F;
    F.function = &f;
    F.params = &alpha; // automatic cast to void*

    const double int_min = 0.0, int_max = 1.0;
    const double rel_error = 1e-7, abs_error = 0.0;

    gsl_integration_qags (&F, int_min, int_max, abs_error,
        rel_error, max_num_subdiv, w, &result, &error);

    printf ("result      = % .18f\n", result);
    printf ("exact result = % .18f\n", -4.0);
    printf ("estimated error = % .18f\n", error);
    printf ("actual error   = % .18f\n", result - (-4.0));

    gsl_integration_workspace_free (w);
    return 0;
}
```

GSL

Integration: adaptive

Flexibility mandates that the **params ptr** on **gsl\_function** to be a **void** **void pointer** is the base of generic programming in C

Any pointer can be casted into the **void pointer**.

The converse is also true as a **void pointer** can be casted into any pointer

```
gsl_integration_workspace *
gsl_integration_workspace_alloc (const size_t n)
{
    gsl_integration_workspace * w ;
    if (n == 0)
    {
        GSL_ERROR_VAL ("workspace length n must be positive integer",
                      GSL_EDOM, 0);
    }

    w = (gsl_integration_workspace *) malloc (sizeof (gsl_integration_workspace));
    if (w == 0)
    {
        GSL_ERROR_VAL ("failed to allocate space for workspace struct",
                      GSL_ENOMEM, 0);
    }

    w->alist = (double *) malloc (n * sizeof (double));
    if (w->alist == 0)
    {
        free (w);          /* exception in constructor, avoid memory leak */
        GSL_ERROR_VAL ("failed to allocate space for alist ranges",
                      GSL_ENOMEM, 0);
    }

    w->blist = (double *) malloc (n * sizeof (double));
    if (w->blist == 0)
    {
        free (w->alist);
        free (w);          /* exception in constructor, avoid memory leak */
        GSL_ERROR_VAL ("failed to allocate space for blist ranges",
                      GSL_ENOMEM, 0);
    }

    w->rlist = (double *) malloc (n * sizeof (double));
    if (w->rlist == 0)
    {
```

## Why some GSL functions need workspaces?

To store intermediate results

from GSL source code

```
typedef struct
{
    size_t limit;
    size_t size;
    size_t nrmax;
    size_t i;
    size_t maximum_level;
    double *alist;
    double *blist;
    double *rlist;
    double *elist;
    size_t *order;
    size_t *level;
}
gsl_integration_workspace;
```

```

#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double f (double x, void* params)
{
    double alpha = *(double *) params;
    return alpha*x*x;
}

int main (void)
{
    size_t num_subdiv;
    const double int_min = 0.0, int_max = 1.0;
    const double rel_error = 1e-3, abs_error = 0.0;
    double result, error, alpha = 1.0;

    gsl_function F;
    F.function = &f;
    F.params = &alpha; // automatic cast to void*

    gsl_integration_qng (&F, int_min, int_max, abs_error,
                         rel_error, &result, &error, &num_subdiv);

    printf ("result          = % .18f\n", result);
    printf ("exact result    = % .18f\n", 1.0/3.0);
    printf ("estimated error = % .18f\n", error);
    printf ("actual error    = % .18f\n", result - (1./3.));

    return 0;
}

```

## GSL Integration: non-adaptive

```

whovian@phy504:~/host/C$ gcc -O2 gsl3.c -o gsl3 -lgsl -lgslcblas -lm
whovian@phy504:~/host/C$ ./gsl3
result          = 0.3333333333333370
exact result    = 0.3333333333333315
estimated error = 0.000000000000003701
actual error    = 0.00000000000000056

```

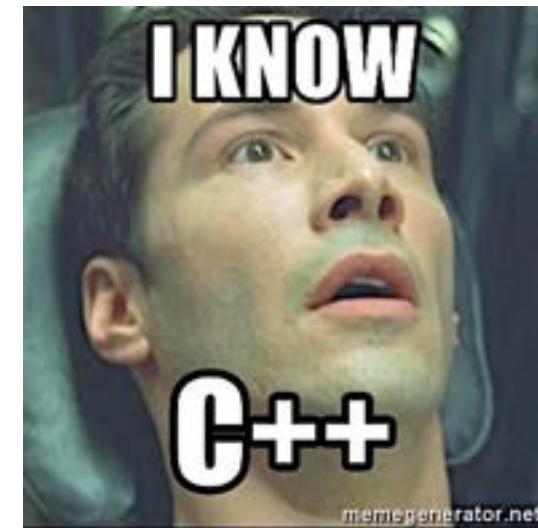
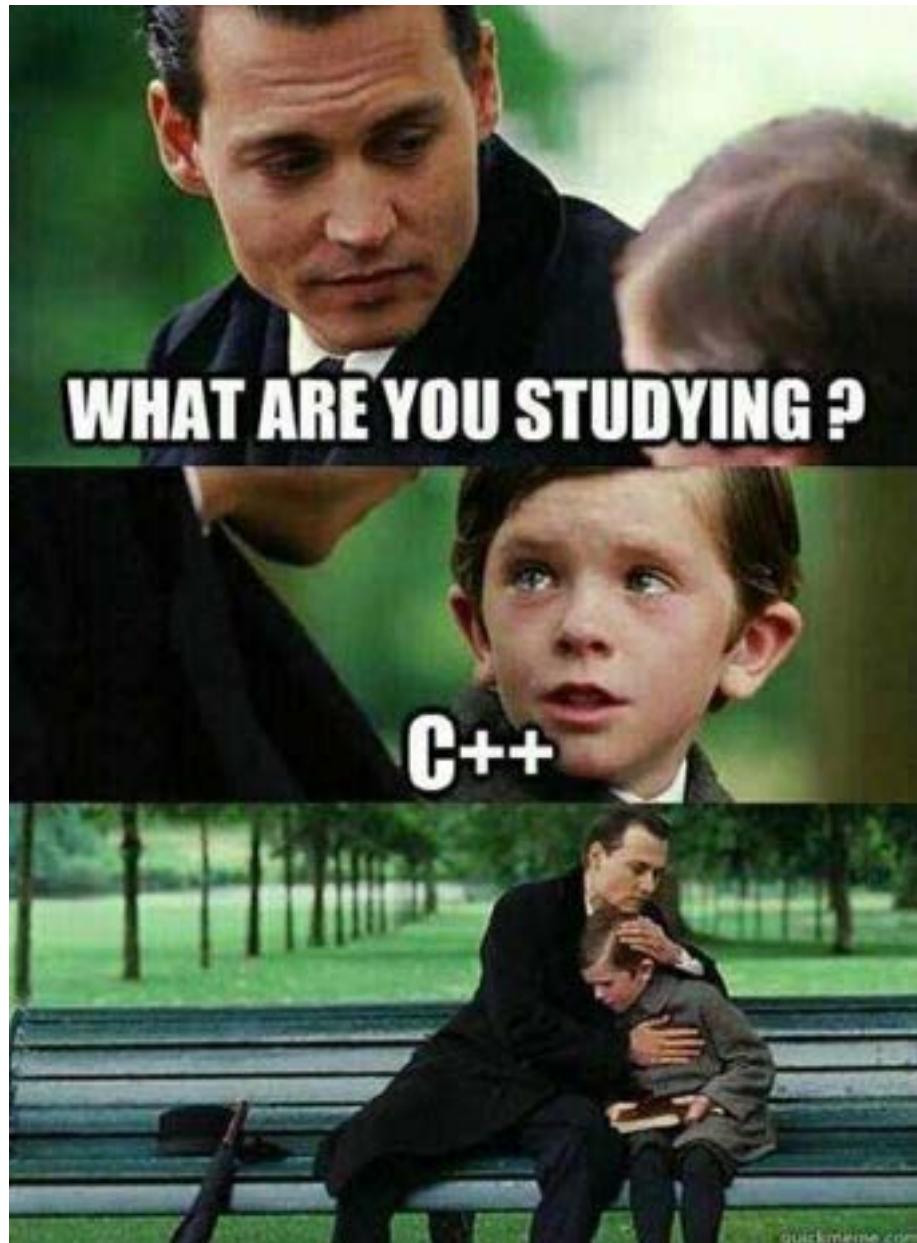
- **QNG:** weaker, non-adaptive but fast integrator for smooth functions

Flexibility mandates that params on **gsl\_function** is a void pointer.

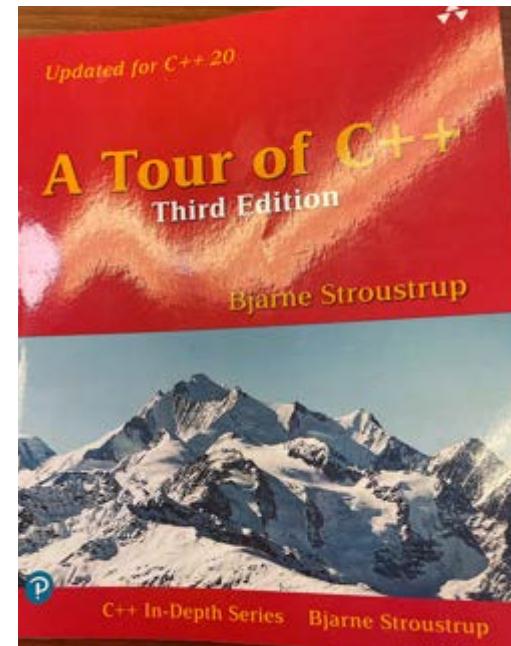
Source:

<https://www.gnu.org/software/gsl/doc/html/integration.html>

# Lecture 25: C++ part I



## Suggested Literature



# Welcome to C++. Important Advice

C++ allowed on google code (google style)			cutting edge C++		
C++ 93 C 93	C++ 2011 2011	C++ 2013 2013	C++ 2017 2017	C++ 2020 2020	C++ 2023 2023
gcc full support	gcc 4.8	gcc 5.0	gcc 8.0	gcc 12.0	partial

← MOSTLY BACKWARD COMPATIBLE

C++ is a huge language that is constantly evolving (new standard every three years)

Not all features available by the latest standard are available yet

C++ file extensions: **cpp** and **hpp** (header). I also use **.c** and **.h** (old habits die hard)

# Welcome to C++. Important Advice

C++ allowed on google code (google style)			cutting edge C++		
C++ 93	C++ 2011	C++ 2013	C++ 2017	C++ 2020	C++ 2023
gcc full support	gcc 4.8	gcc 5.0	gcc 8.0	gcc 12.0	partial
← MOSTLY BACKWARD COMPATIBLE					

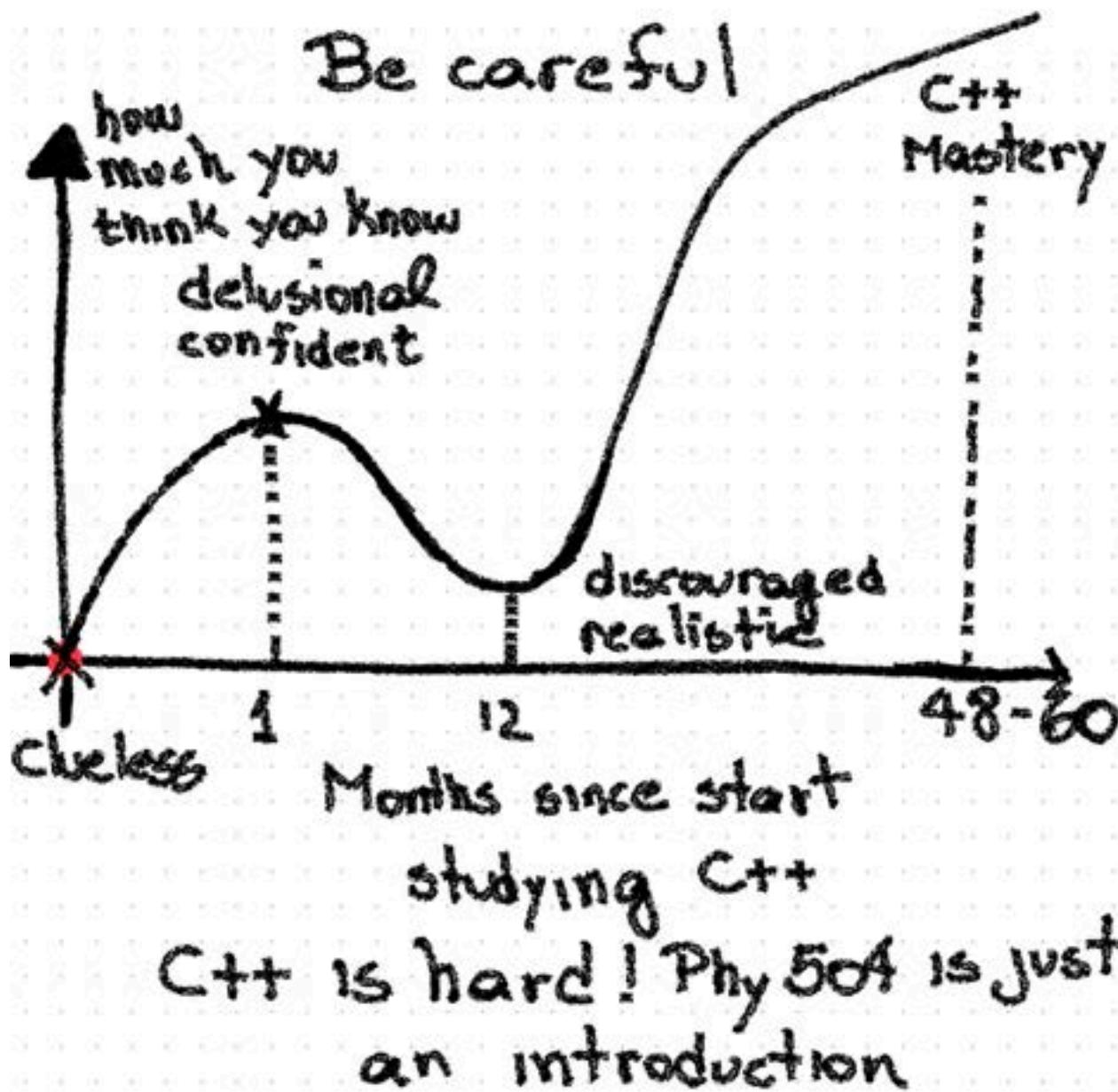
You need to be careful about compiler version

We are going to use **gcc** version **12**

Not available at the original docker Phy504 image (so please reinstall it)

Use the compiler flag **-std=c++XX** to choose the standard

# Welcome to C++. Important Advice



No need to learn all C++ features at once to do research.

I love C++, but it takes a long time to master it.

[Google C++ style](#) is a good conservative choice of features

# Google C++ Style

## Good Survival Guide: we won't respect it during class (but still a good idea to follow it)

Header Files	<a href="#">Self-contained Headers</a> <a href="#">The #define Guard</a> <a href="#">Include What You Use</a> <a href="#">Forward Declarations</a> <a href="#">Inline Functions</a> <a href="#">Names and Order of Includes</a>
Scoping	<a href="#">Namespaces</a> <a href="#">Internal Linkage</a> <a href="#">Nonmember, Static Member, and Global Functions</a> <a href="#">Local Variables</a> <a href="#">Static and Global Variables</a> <a href="#">thread_local Variables</a>
Classes	<a href="#">Doing Work in Constructors</a> <a href="#">Implicit Conversions</a> <a href="#">Copyable and Movable Types</a> <a href="#">Structs vs. Classes</a> <a href="#">Structs vs. Pairs and Tuples</a> <a href="#">Inheritance</a> <a href="#">Operator Overloading</a> <a href="#">A</a>
Functions	<a href="#">Inputs and Outputs</a> <a href="#">Write Short Functions</a> <a href="#">Function Overloading</a> <a href="#">Default Arguments</a> <a href="#">Trailing Return Type Syntax</a>
Google-Specific Magic	<a href="#">Ownership and Smart Pointers</a> <a href="#">cpplint</a>
Other C++ Features	<a href="#">Rvalue References</a> <a href="#">Friends</a> <a href="#">Exceptions</a> <a href="#">noexcept</a> <a href="#">Run-Time Type Information (RTTI)</a> <a href="#">Casting</a> <a href="#">Streams</a> <a href="#">Preincrement and Predecrement</a> <a href="#">Use of const</a> <a href="#">Use of con</a> <a href="#">0 and nullptr/NULL</a> <a href="#">sizeof</a> <a href="#">Type Deduction (including auto)</a> <a href="#">Class Template Argument Deduction</a> <a href="#">Designated Initializers</a> <a href="#">Lambda Expressions</a> <a href="#">Template Metaprogrammi</a>
Inclusive Language	
Naming	<a href="#">General Naming Rules</a> <a href="#">File Names</a> <a href="#">Type Names</a> <a href="#">Variable Names</a> <a href="#">Constant Names</a> <a href="#">Function Names</a> <a href="#">Namespace Names</a> <a href="#">Enumerator Names</a> <a href="#">Macro Names</a> <a href="#">Exc</a>
Comments	<a href="#">Comment Style</a> <a href="#">File Comments</a> <a href="#">Class Comments</a> <a href="#">Function Comments</a> <a href="#">Variable Comments</a> <a href="#">Implementation Comments</a> <a href="#">Punctuation, Spelling, and Grammar</a> <a href="#">TODO C</a>
Formatting	<a href="#">Line Length</a> <a href="#">Non-ASCII Characters</a> <a href="#">Spaces vs. Tabs</a> <a href="#">Function Declarations and Definitions</a> <a href="#">Lambda Expressions</a> <a href="#">Floating-point Literals</a> <a href="#">Function Calls</a> <a href="#">Braced Initiali</a> <a href="#">Pointer and Reference Expressions</a> <a href="#">Boolean Expressions</a> <a href="#">Return Values</a> <a href="#">Variable and Array Initialization</a> <a href="#">Preprocessor Directives</a> <a href="#">Class Format</a> <a href="#">Constructor Initializer L</a>
Exceptions to the Rules	<a href="#">Existing Non-conformant Code</a> <a href="#">Windows Code</a>

# Hello World

Many things to discuss from a simple Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;

    std::cout << "Hello World\n";

    return 0;
}

whovian@phy504:~/host/CPP$ g++ hello.cpp -o hello1 --std=c++20
whovian@phy504:~/host/CPP$ ./hello1
Hello World
Hello World
Hello World
```

# Hello World

Many things to discuss from a simple Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below
int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;

    std::cout << "Hello World\n";

    return 0;
}
```

no .h at the end  
of the header

```
whovian@phy504:~/host/CPP$ g++ hello.cpp -o hello1 --std=c++20
whovian@phy504:~/host/CPP$ ./hello1
Hello World
Hello World
Hello World
```

# Hello World

Many things to discuss from a simple Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;

    std::cout << "Hello World\n";

    return 0;
}
```

With **printf**, the code offloads the string to the terminal after every command

When printing a lot of stuff that is slow and C++ is crazy about optimization

# Hello World

Many ways to write Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;
    std::cout << "Hello World\n";

    return 0;
}
```

C++ **std::cout**  
has an  
intermediate  
**buffer**

**std::endl = “\n”**  
+ flush buffer  
**now**

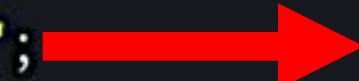
# Hello World

Many things to discuss from a simple Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;

    std::cout << "Hello World\n"; 
}

return 0;
```

C++ **std::cout**  
has an  
intermediate  
**buffer**

**std::endl = “\n”**  
+ flush buffer  
**now** (safer but  
slower - speed  
only relevant on  
extreme cases)

# Hello World

What is **std::**? C++ namespaces

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");
    std::cout << "Hello World" << std::endl;
    std::cout << "Hello World\n";
    return 0;
}
```

It is ok to use  
**using namespace std**  
in **cpp** files

Not good  
practice to use it  
in **hpp** (headers).

This would  
destroy the  
encapsulation  
purpose of  
namespaces

# Hello World

What is **std::**? C++ namespaces

```
#include <iostream> // C++17 and below  
  
using namespace std; // make std namespace  
// global  
  
int main ()  
{  
    cout << "Hello World" << endl;  
  
    cout << "Hello World\n";  
  
    return 0;  
}
```

It is ok to use  
**using**  
**namespace std**  
in **cpp** files

Not good  
practice to use it  
in **hpp** (headers).

This would  
destroy the  
encapsulation  
purpose of  
namespaces

# Hello World

Informal rule: C++ offers >1 ways to solve problems

```
#include <fmt/core.h>
// FMT: C++20 implementation of std::format
// FMT: C++23 implementation of std::print
// (not implemented in gcc yet)

int main ()
{
    fmt::print("Hello World!\n");

    // Positional Arguments
    fmt::print("{1} {0}.\n", "World", "Hello");

    fmt::print(
        "Hello, {name}! The answer is {number}. Goodbye, {name}\n",
        fmt::arg("name", "World"), fmt::arg("number", 42));
}
```

```
whovian@phy504:~/host$ nano hello3.cpp
whovian@phy504:~/host$ g++ hello3.cpp -o hello3 -std=c++20 -lfmt
whovian@phy504:~/host$ ./hello3
Hello World!
Hello World.
Hello, World! The answer is 42. Goodbye, World
```

C++23  
introduces  
**std::print** (not  
implemented  
on **gcc** yet)

Standard based  
on the  
FMT library

```
#include <iostream>
#include <fmt/core.h>
// FMT: C++20 implementation of std::format
// FMT: C++23 implementation of std::print
// (not implemented in gcc yet)

int main ()
{
    int x = 2;
    // this is NOT an array. It is int initialization
    int y{3};

    // init an integer from std::initializer_list<int>
    // the type of {3} is std::initializer_list<int>
    int z = {3};

    int w{3};

    // not formated output (basic way since C++<11)
    std::cout << "Ints are " << x << ", " << y << ", " << z << " and "
        << w << std::endl;

    // Second way of printing: formated (C++20)
    std::cout <<
        fmt::format("Ints are {:1d}, {:1d}, {:1d} and {:1d}", x, y, z, w)
        << std::endl;

    // Third way of printing: formated w/ std::print (C++23)
    fmt::print("Ints are {:1d}, {:1d}, {:1d} and {:1d}\n", x, y, z, w);
}
```



```
whovian@phy504:~/host$ g++ init.cpp -o init -std=c++20 -lfmt
whovian@phy504:~/host$ ./init
Ints are 2, 3, 3 and 3
Ints are 2, 3, 3 and 3
Ints are 2, 3, 3 and 3
```

# Variable initialization

There are >1 way to initialize even basic types in C++

Advanced Topic: they have slightly different behaviors

```
#include <iostream>
#include <fmt/core.h>
// FMT: C++20 implementation of std::format
// FMT: C++23 implementation of std::print
// (not implemented in gcc yet)

int main ()
{
    int x = 2;
    // this is NOT an array. It is int initialization
    int y{3};

    // init an integer from std::initializer_list<int>
    // the type of {3} is std::initializer_list<int>
    int z = {3};

    int w{3};

    // not formated output (basic way since C++<11)
    std::cout << "Ints are " << x << ", " << y << ", " << z << " and "
        << w << std::endl;

    // Second way of printing: formated (C++20)
    std::cout <<
        fmt::format("Ints are {:1d}, {:1d}, {:1d} and {:1d}", x, y, z, w)
        << std::endl;

    // Third way of printing: formated w/ std::print (C++23)
    fmt::print("Ints are {:1d}, {:1d}, {:1d} and {:1d}\n", x, y, z, w);
}
```

Variable  
initialization

Example of  
standard and  
bleeding-edge  
(C++23)  
ways to print  
variables

Two  
commands:  
**print** and  
**format**

```
#include <string>
#include <iostream>

namespace vivian
{
    void Print(double x)
    {
        std::cout << "Namespace Vivian\n";
        std::cout << "We will print a double. x = " <<
            x << std::endl;
    }
}

namespace phy504
{
    void Print(double x)
    {
        std::cout << "Namespace Phy504" << std::endl;
        std::cout << "We will print a double. x = " <<
            x << std::endl;
    }
}

int main()
{
    vivian::Print(2.0);
    phy504::Print(3.0);
    return 0;
}
```

# Namespaces

Namespaces allow  
duplication of  
function names with  
same type / num args

Writing  
**using namespace std**  
pollutes your code  
(acceptable on **cpp**)

```
whovian@phy504:~/host/CPP$ nano functions4.cpp
whovian@phy504:~/host/CPP$ g++ functions4.cpp -o func4 -std=c++20
whovian@phy504:~/host/CPP$ ./func4
Namespace Vivian
We will print a double. x = 2
Namespace Phy504
We will print a double. x = 3
```

```
#include <string>
#include <iostream>

namespace vivian
{
    int myglobal = 10;

    void Print(double x)
    {
        std::cout << "Namespace Vivian\n";
        std::cout << "We will print a double. x = " <<
            x << std::endl;
    }
}

namespace phy504
{
    int myglobal = 20;

    void Print(double x)
    {
        std::cout << "Namespace Phy504" << std::endl;
        std::cout << "We will print a double. x = " <<
            x << std::endl;
    }
}

int main()
{
    std::cout << vivian::myglobal << " " <<
        phy504::myglobal << std::endl;
    return 0;
}
```

# Namespaces

You can also declare global variables inside **namespaces** (provides a minimum level of encapsulation)

Writing  
**using namespace std**  
pollutes your code

```
whovian@phy504:~/host/CPP$ nano functions5.cpp
whovian@phy504:~/host/CPP$ g++ functions5.cpp -o func5 -std=c++20
whovian@phy504:~/host/CPP$ ./func5
```

Ok on **cpp** files, terrible  
on **hpp** (headers)

```
#include <string>
#include <iostream>

void Print(double x)
{
    std::cout << "We will print a double. x = " <<
        x << std::endl;
}
void Print(int x)
{
    std::cout << "We will print an int. x = " <<
        x << std::endl;
}
void Print(std::string x)
{
    std::cout << "We will print a C++ string " <<
        x << std::endl;
}

int main()
{
    Print(1);
    Print(2.0);

    double x = 3.0;
    Print(3.0);

    std::string str = "HelloWorld";
    Print(str);
    Print("What?");

    return 0;
}
```

# Functions C vs C++

Functions in C++ allow naming overloading

C++ lets you specify >1 function w/ same name in the same scope

Overloading works on types

Argument order is imp.

**f(int, double)** and **f(double, int)** are different

```
whovian@phy504:~/host/CPP$ g++ -std=c++20 functions.cpp -o func
whovian@phy504:~/host/CPP$ ./func
We will print an int. x = 1
We will print a double. x = 2
We will print a double. x = 3
We will print a C++ string HelloWorld
We will print a C++ string What?
```

```
#include <string>
#include <iostream>

void Print(int x, double y)
{
    std::cout << "Print1: " <<
        x+y << std::endl;
}

void Print(double x, int y)
{
    std::cout << "Print2: " <<
        x-y << std::endl;
}

int main()
{
    Print(1, 2.0);
    Print(1.0, 2);
    return 0;
}
```

## Functions C vs C++

Functions in C++ allow naming overloading

C++ lets you specify >1 function w/ same name in the same scope

Overloading works on types Argument order is imp. **f(int, double)** and **f(double, int)** are different

```
whovian@phy504:~/host/CPP$ g++ -std=c++20 functions2.cpp -o func2
whovian@phy504:~/host/CPP$ ./func2
Print1: 3
Print2: -1
```

```
#include <string>
#include <iostream>

void Print(int x, double y)
{
    std::cout << "Print1: " <<
        x+y << std::endl;
}

void Print(double x, int y)
{
    std::cout << "Print2: " <<
        x-y << std::endl;
}

int main()
{
/*
    Print(1, 1); // error - compiler can't
                  // decide which function
                  // to use
*/
    // We need to do explicit casting
    // C++ static_cast
    Print(static_cast<double>(1), 1);
    Print(1, static_cast<double>(1));
    return 0;
}
```

# Functions C vs C++

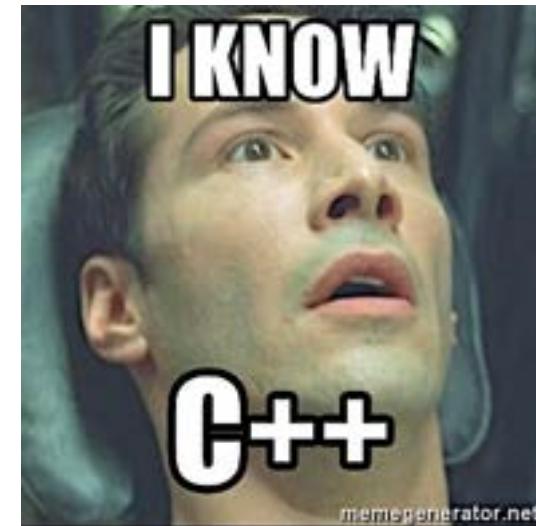
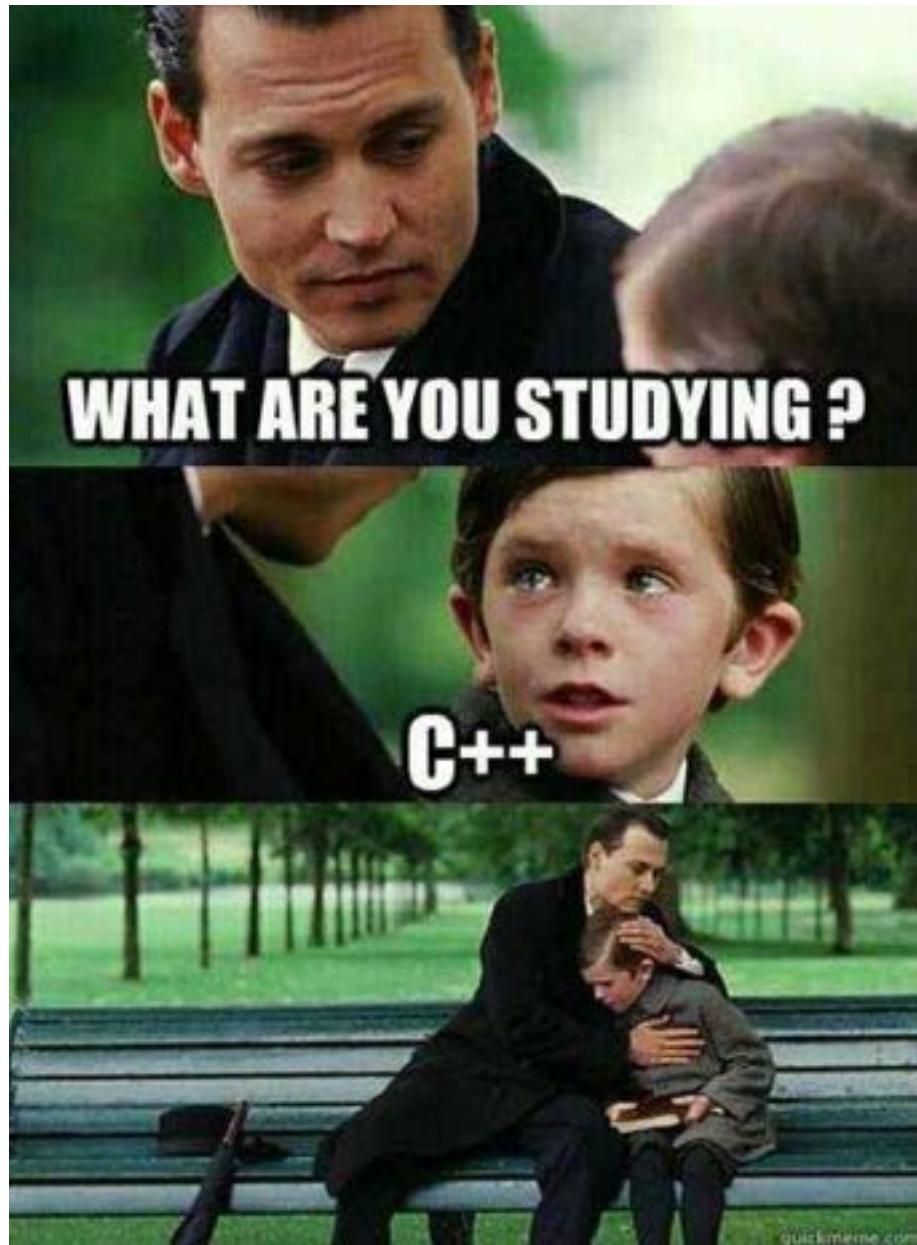
Functions in C++ allow naming overloading

C++ lets you specify >1 function w/ same name in the same scope

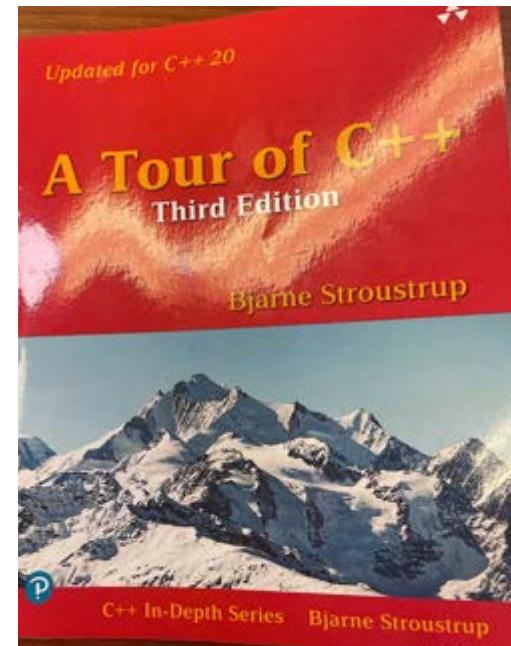
Overloading works on types Argument order is imp. **f(int, double)** and **f(double, int)** are different

```
whovian@phy504:~/host/CPP$ g++ -std=c++20 functions3.cpp -o func3
whovian@phy504:~/host/CPP$ ./func3
Print2: 0
Print1: 2
```

# Lecture 26: C++ part II



## Suggested Literature



```
#include <tuple>
#include <fmt/core.h>

int main()
{
    std::tuple<std::string, int> x =
        std::make_tuple("Vivian Miranda", 37);

    // Access tuple elements by position
    fmt::print("name = {0}. Age = {1}\n",
              std::get<0>(x), std::get<1>(x));

    // Access tuple elements by type
    // As long as the type of elements are diff
    fmt::print("name = {0}. Age = {1}\n",
              std::get<std::string>(x), std::get<int>(x));

    // If you only have 3 elements, use std::pair
    // std::pair has nice syntax to access elements
    std::pair<std::string, int> y =
        std::make_pair("Vivian Miranda", 37);

    fmt::print("name = {0}. Age = {1}\n",
              y.first, y.second);
}

return 0;
}
```

whovian@phy504:~/host\$ g++ tuples.cpp -o tuple -std=c++20 -lfmt  
whovian@phy504:~/host\$ ./tuple  
name = Vivian Miranda. Age = 37  
name = Vivian Miranda. Age = 37  
name = Vivian Miranda. Age = 37

# Functions

## C vs C++

Functions + STL  
allows the return of  
multiple objects.  
How?

### C++ **tuples** and **pairs** (C++11)

### **tuples / pairs**

allow elements  
with different types  
to be stored

```
#include <tuple>
#include <fmt/core.h>

int main()
{
    // tuple types are too long
    //std::tuple<std::string, int> x =
    //    std::make_tuple("Vivian Miranda", 37);

    // Modern C++ has a powerful tool to
    // facilitate the use of complicated types

    // compiler can figure out the type based on RHS!
    auto x = std::make_tuple("Vivian Miranda", 37);

    // Access tuple elements by position
    fmt::print("name = {0}. Age = {1}\n",
              std::get<0>(x), std::get<1>(x));

    auto y = std::make_pair("Vivian Miranda", 37);

    fmt::print("name = {0}. Age = {1}\n",
              y.first, y.second);

    return 0;
}
```

whovian@phy504:~/host\$ nano tuples2.cpp  
whovian@phy504:~/host\$ ./tuple2  
name = Vivian Miranda. Age = 37  
name = Vivian Miranda. Age = 37

## Functions C vs C++

Need a special function to make a  
**tuple / pair**  
**(make\_tuple/pair)**

You **cannot** access tuple elements via  
**[] operator**. We need  
**std::get<index>**  
or **std::get<type>**

```
#include <fmt/core.h>
#include <tuple>
#include <string>

// example from en.cppreference.com/w/cpp/utility/tuple

std::tuple<double, char, std::string>
get_student(const int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
        case 1:
            return {2.9, 'C', "Milhouse Van Houten"};
        case 2:
            return {1.7, 'D', "Ralph Wiggum"};
    }

    // C++ error handling (we will revisit them later
    throw std::invalid_argument("id");
}

int main()
{ // Many ways to access the return type of get_students
    // 1: with tuples
    //     Yes, we can apply const correctness to auto!
    const auto student0 = get_student(0);

    fmt::print("ID: {0}, GPA: {1}, grade: {2}, name: {3}\n",
              0,
              std::get<double>(student0),
              std::get<char>(student0),
              std::get<std::string>(student0));

    fmt::print("ID: {0}, GPA: {1}, grade: {2}, name: {3}\n",
              0,
              std::get<0>(student0), whovian@phy504:~/host$ g++ tuples3.cpp -o tuple3 -std=c++20 -lfmt
              std::get<1>(student0),
              std::get<2>(student0));
    return 0;
}
```

# Functions C vs C++

We need **std::get<index>** or **std::get<type>** to access tuple elements. How?

**std::get<index>** is never ambiguous. On the other hand, **std::get<type>** can only be used when all types are different

```
whovian@phy504:~/host$ ./tuple3
ID: 0, GPA: 3.8, grade: A, name: Lisa Simpson
ID: 0, GPA: 3.8, grade: A, name: Lisa Simpson
```

```
#include <fmt/core.h>
#include <tuple>
#include <string>

// example from en.cppreference.com/w/cpp/utility/tuple

auto get_student(const int id)
{
    switch (id)
    {
        case 0:
            return std::make_tuple(3.8, 'A', "Lisa Simpson");
        case 1:
            return std::make_tuple(2.9, 'C', "Milhouse Van Houten");
        case 2:
            return std::make_tuple(1.7, 'D', "Ralph Wiggum");
    }

    throw std::invalid_argument("id");
}

int main()
{
    const auto student0 = get_student(0);

    fmt::print("ID: {0}, GPA: {1}, grade: {2}, name: {3}\n",
              0,
              std::get<0>(student0),
              std::get<1>(student0),
              std::get<2>(student0));

    return 0;
}
```

whovian@phy504:~/host/Phy504Docker/CPP\$ g++ tuples3v2.cpp -o tuples3v2 -std=c++20 -lfmt  
whovian@phy504:~/host/Phy504Docker/CPP\$ ./tuples3v2  
ID: 0, GPA: 3.8, grade: A, name: Lisa Simpson

## Functions C vs C++

Can we adapt the example from the previous slide to return auto?

Yes!

But in this case, we need to create the tuples with  
**std::make\_tuple**

```
#include <fmt/core.h>
#include <tuple>
#include <string>

// example from en.cppreference.com/w/cpp/utility/tuple

std::tuple<double, char, std::string>
get_student(const int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
        case 1:
            return {2.9, 'C', "Milhouse Van Houten"};
        case 2:
            return {1.7, 'D', "Ralph Wiggum"};
    }

    // C++ error handling (we will revisit them later
    throw std::invalid_argument("id");
}

int main()
{ // Many ways to access the return type of get_students

    // 2: with tie
    // a bit clunky because you need to define vars separately
    double gpa;
    char grade;
    std::string name;

    std::tie(gpa, grade, name) = get_student(2);
    fmt::print("ID: {}, GPA: {}, grade: {}, name: {}\n",
              2, gpa, grade, name);
    return 0;
}
```

# Functions C vs C++

What if you don't want  
to create a tuple to  
receive multiple returns?

First option: **std::tie**  
(C++11). Great, except  
that you can't use it with  
the **auto** keyword (need  
to write the variable  
types explicitly)

```
whovian@phy504:~/host$ g++ tuples4.cpp -o tuple4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple4
ID: 2, GPA: 1.7, grade: D, name: Ralph Wiggum
```

```

#include <fmt/core.h>
#include <tuple>
#include <string>

// example from en.cppreference.com/w/cpp/utility/tuple

std::tuple<double, char, std::string>
get_student(const int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
        case 1:
            return {2.9, 'C', "Milhouse Van Houten"};
        case 2:
            return {1.7, 'D', "Ralph Wiggum"};
    }

    // C++ error handling (we will revisit them later
    throw std::invalid_argument("id");
}

int main()
{ // Many ways to access the return type of get_students

    // 3: (Best) with C++17 structured binding:
    const auto [ gpa, grade, name ] = get_student(1);

    fmt::print("ID: {}, GPA: {}, grade: {}, name: {}\n",
              1, gpa, grade, name)
    return 0;
}

```

# Functions C vs C++

What if you don't want to create a tuple to receive multiple returns?

Second option: with C++17 **structured binding** (allows the use of **auto**, types can be deducted)

```

whovian@phy504:~/host$ g++ tuples5.cpp -o tuple5 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple5
ID: 1, GPA: 2.9, grade: C, name: Milhouse Van Houten

```

```
#include <fmt/core.h>
#include <tuple>
#include <string>

// example from en.cppreference.com/w/cpp/utility/tuple

std::tuple<double, char, std::string>
get_student(const int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
        case 1:
            return {2.9, 'C', "Milhouse Van Houten"};
    }

    throw std::invalid_argument("id");
}

int main()
{
    const auto [ gpa, grade, name ] = get_student(3);

    // this part of the program won't run
    fmt::print("ID: {}, GPA: {}, grade: {}, name: {}\n",
              1, gpa, grade, name);

    return 0;
}
whovian@phy504:~/host$ g++ tuples6.cpp -o tuple6 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple6
terminate called after throwing an instance of 'std::invalid_argument'
  what():  id
qemu: uncaught target signal 6 (Aborted) - core dumped
Aborted
```

# Exceptions

What happens if we type  
the wrong id?

C++ allows the use of  
exceptions to do error  
handling

Exception Safety is an  
advanced topic. Rule for  
newbies: dangerous to use

Why? Because it disrupts  
the normal flow of the  
program (what if we have  
an array in the heap?)

```
#include <fmt/core.h>
#include <tuple>
#include <string>

// example from en.cppreference.com/w/cpp/utility/tuple
std::tuple<double, char, std::string> get_student(int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
    }
    std::string emsg =
        fmt::format("id error on function {}", __FUNCTION__);
    throw std::invalid_argument(emsg);

    fmt::print("This won't run!!\n");
}
int main()
{
    // catching exceptions: try/catch makes code clucky
    // usually I only try to catch exceptions in the main()
    // function, so I can write some extra debugging msg

    try {
        const auto [ gpa, grade, name ] = get_student(3);
        // code below won't run
        fmt::print("ID: {}, GPA: {}, grade: {}, name: {}\n",
                  1, gpa, grade, name);
    }
    catch (std::exception& ex) {
        fmt::print("Error (file: {}, line: {})\n", __FILE__, __LINE__);
        fmt::print("{}\n", ex.what());
        // problem - fmt::print itself may throw an exception!
        // (yes - exception is difficult to get 100% right)
        exit(1);
    }
    catch (...) {
        fmt::print("Error (file: {}, line: {})\n",
                  __FILE__, __LINE__);
        throw std::runtime_error("Unknown Exception\n");
    }
    return 0;
}
```

# Exceptions

Exception: dangerous

Why? Because it disrupts  
the normal flow of the  
program

So, if you use **malloc / free** (or the C++ version **new / delete**), then the C++ exceptions can create memory leaks

```
whovian@phy504:~/host$ g++ tuples7.cpp -o tuple7 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple7
Error (file: tuples7.cpp, line: 32)
id error on function get_student
```

# Functions C vs C++

```
#include <fmt/core.h>

// (1) Does the optional arg have a logical default?
// (2) Is the function implemented the same when the
//     optional ar is given and when it's not?
// From SO: stackoverflow.com/a/56388368/2472169
// if YES and YES, use default argument
// if NO and YES, then use std::optional<T>
// if NO and NO, then use overloading

using fptr = int (*)(int, int);

int add(int x, int y, fptr sum = nullptr) {
    return (sum == nullptr) ? x + y : sum(x, y);
}

int sum(int x, int y) { return x + y; }

int main()
{
    fmt::print("Default arg: {}\n", add(2, 3));
    fmt::print("Default arg: {}\n", add(2, 3, &sum));
}
```

Functions can have  
optional arguments and  
return values

How?

1. default values
2. overloading
3. std::optional<T>

```
whovian@phy504:~/host$ g++ optional3.cpp -o opt3 -std=c++20 -lfmt
whovian@phy504:~/host$ ./opt3
Default arg: 5
Default arg: 5
```

```
#include <fmt/core.h>

// (1) Does the optional arg have a logical default?
// (2) Is the function implemented the same when the
//     optional ar is given and when it's not?
// From SO: stackoverflow.com/a/56388368/2472169
// if YES and YES, use default argument
// if NO and YES, then use std::optional<T>
// if NO and NO, then use overloading

using fptr = int (*)(int, int);

int add(int x, int y) { // here we avoid if/else struct
    return x + y;
}
int add(int x, int y, fptr sum) {
    return sum(x, y);
}
int add(int x, int y, std::nullptr_t) {
    // nullptr_t = type of nullptr (C++ version of NULL)
    return x + y;
}

int sum(int x, int y) { return x + y; }

int main()
{
    fmt::print("Overloading: {} \n", add(2, 3));
    fmt::print("Overloading: {} \n", add(2, 3, &sum));
    fmt::print("Overloading: {} \n", add(2, 3, nullptr));
}
```

# Functions C vs C++

Functions can have  
optional arguments and  
return values

How?

1. default values
2. overloading
3. std::optional<T>

```
whovian@phy504:~/host$ g++ optional4.cpp -o opt4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./opt4
Overloading: 5
Overloading: 5
Overloading: 5
```

```

#include <cstddef>
#include <optional>
#include <fmt/core.h>

// (1) Does the optional arg have a logical default?
// (2) Is the function implemented the same when the
//     optional arg is given and when it's not?
// From SO: stackoverflow.com/a/56388368/2472169
// if YES and YES, use default argument
// if NO and YES, then use std::optional<T>
// if NO and NO, then use overloading

using fptr = int (*)(int, int);

int add(int x, int y, std::optional<fptr> sum) {
    return (sum.has_value() == 0 || sum == nullptr) ?
        x + y : sum.value()(x,y);
}

int sum(int x, int y) { return x + y; }

int main()
{
    // interesting here - you must always give 3 args
    fmt::print("std::optional: {}\n", add(2, 3, nullptr));
    fmt::print("std::optional: {}\n", add(2, 3, {}));
    fmt::print("std::optional: {}\n", add(2, 3, std::nullopt));
    fmt::print("std::optional: {}\n", add(2, 3, &sum));
}

```

```

whovian@phy504:~/host$ g++ optional5.cpp -o opt5 -std=c++20 -lfmt
whovian@phy504:~/host$ ./opt5
std::optional: 5
std::optional: 5
std::optional: 5
std::optional: 5

```

# Functions C vs

## C++

Functions can have  
optional arguments  
and return values

How?

1. default values
2. overloading
3. [std::optional<T>](#)

```

#include <cstddef>
#include <optional>
#include <cmath>
#include <fmt/core.h>

double f(double x)
{
    fmt::print("We called double f(double) \n");
    // cbrt = cubic root
    // tgamma = gamma function (C++17 has many special functions!)
    return std::cbrt(x) +
        std::sqrt(std::abs(x)) +
        std::tgamma(x);
}

double f(double* x)
{
    fmt::print("We called double f(double*) \n");
    if (x != nullptr)
        return *x;
    else
        return -1.;
}

int main()
{
    // var type is super important in C++. null.cpp: In function 'int main()':
    double r1 = f(NULL);
    double r2 = f(2.0);
    double r3 = f(nullptr);
}

```

# Functions C vs C++

Why **nullptr** and not **NULL**?

**NULL** is an **int (0)**, but compiler also knows that **NULL** is used with pointers (= confusion)

```

whovian@phy504:~/host$ g++ null.cpp -o null -std=c++20 -lfmt
null.cpp: In function 'int main()':
null.cpp:28:16: error: call of overloaded 'f(NULL)' is ambiguous
      28 |     double r1 = f(NULL);
                  |             ~~~~~
null.cpp:6:8: note: candidate: 'double f(double)'
      6 |     double f(double x)
                  |             ^
null.cpp:16:8: note: candidate: 'double f(double*)'
     16 |     double f(double* x)
                  |             ^

```

```

#include <cstddef>
#include <optional>
#include <cmath>
#include <fmt/core.h>

double f(double x)
{
    fmt::print("We called double f(double) \n");
    // cbrt = cubic root
    // tgamma = gamma function (C++17 has many special functions!)
    return std::cbrt(x) +
        std::sqrt(std::abs(x)) +
        std::tgamma(x);
}

double f(double* x)
{
    fmt::print("We called double f(double*) \n");
    if (x != nullptr)
        return *x;
    else
        return -1.;
}

int main()
{
    // var type is super important in C++. Be careful
    double r2 = f(2.0);
    double r3 = f(nullptr);
}

```

# Functions C vs C++

Why **nullptr** and not  
**NULL**?

**NULL** is an **int (0)**, but  
compiler also knows  
that **NULL** is a ptr!

Don't assume functions  
in C and C++ behave the  
same

```

whovian@phy504:~/host$ g++ null2.cpp -o null2 -std=c++20 -lfmt
whovian@phy504:~/host$ ./null2
We called double f(double)
We called double f(double*)

```

```
#include <fmt/core.h>
#include <spdlog/spdlog.h>
#include <tuple>
#include <string>

std::tuple<double, char, std::string> get_student(int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
    }
    std::string emsg =
        fmt::format("id error on function {}", __FUNCTION__);
    throw std::invalid_argument(emsg);

    fmt::print("This won't run!!");
}

int main()
{
    spdlog::info("Testing spdlog library");
    spdlog::info("Hello, {}!", "World");

    try {
        const auto [ gpa, grade, name ] = get_student(3);
    }
    catch (std::exception& ex) {
        spdlog::critical(ex.what());
        exit(1);
    }
    catch (...) {
        spdlog::critical("Unknown exception");
        exit(1);
    }
    return 0;
}
```

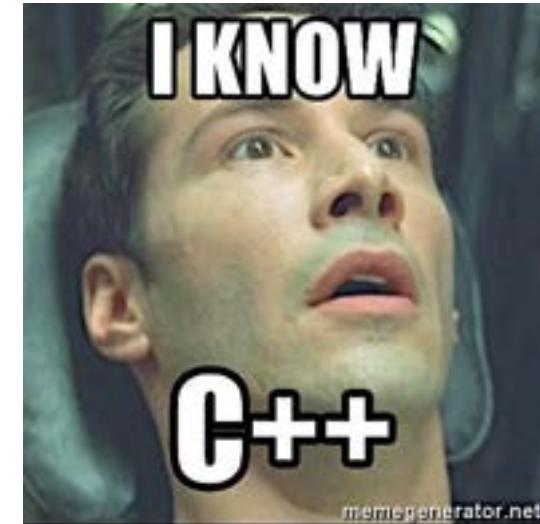
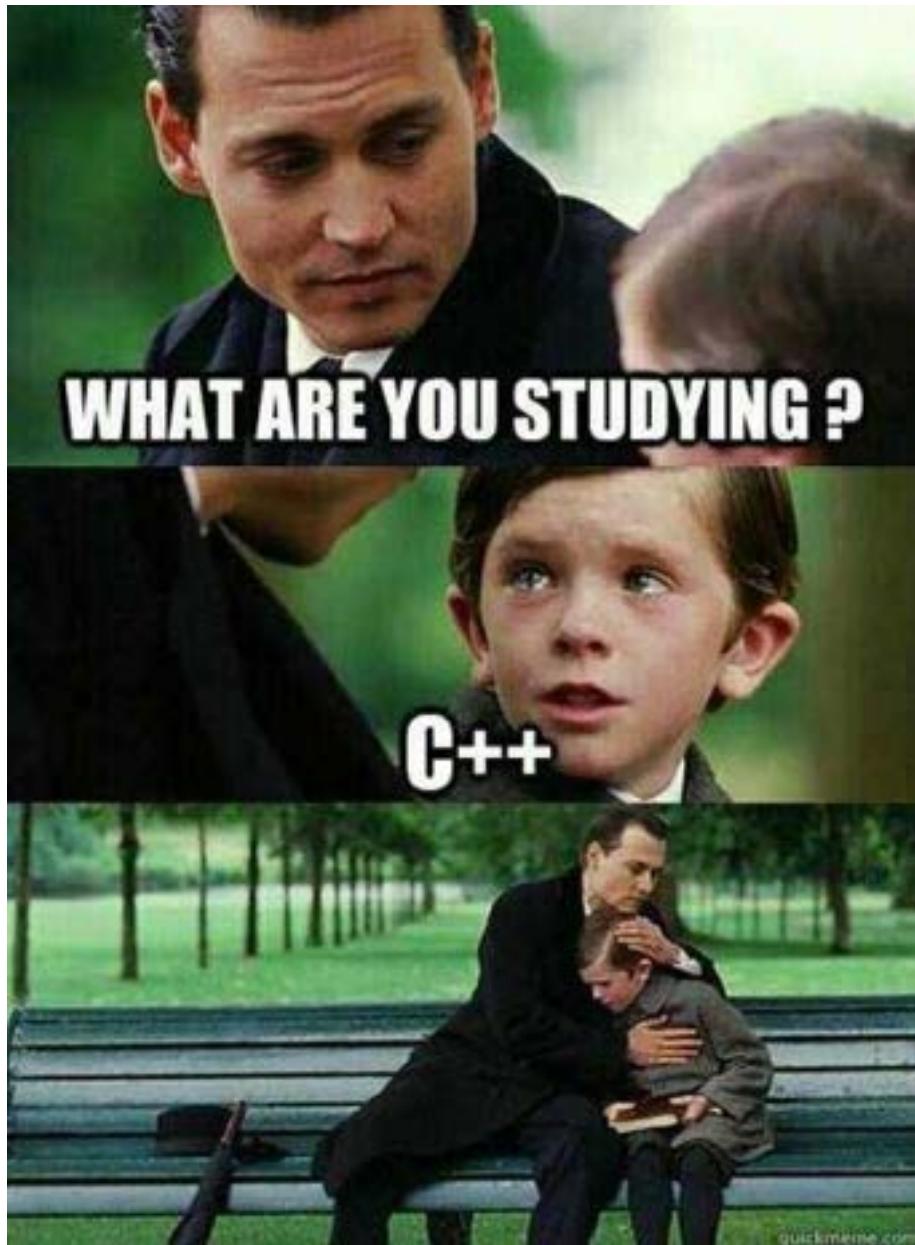
Extra: Logging  
Instead of **fmt::print** /  
**std::cout**, I use a  
professional logging  
library in C++ named  
spdlog

**spdlog** has an option to  
disable exceptions.

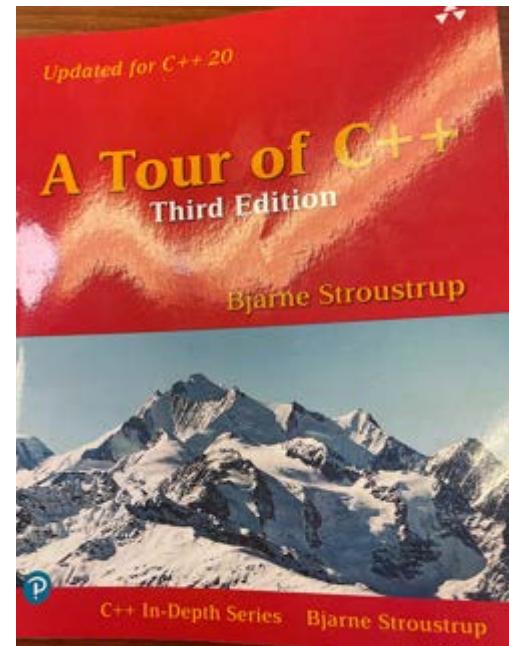
This avoids the  
loophole shown on the  
last slide

```
whovian@phy504:~/host$ g++ tuples8.cpp -o tuple8 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple8
[2023-03-23 22:53:59.008] [info] Testing spdlog library
[2023-03-23 22:53:59.015] [info] Hello, World!
[2023-03-23 22:53:59.018] [critical] id error on function get_student
```

# Lecture 27: C++ part III



## Suggested Literature



```
#include <map>
#include <string>
#include <fmt/core.h>

using std::map;    // better way to avoid using std::
using std::string; // don't pollute your code

int main ()
{
    // map<key type, value type> = initialize list
    // init the map with some key,values
    // We learned that we can use {} to init vars in C++
    map<string, double> mymap = {"Vivian", 37},
                           {"Marcelo", 38},
                           {"Vania", 58},
                           {"Isabella", 25};

    // Similar to Python, we can access some functions with .
    // These are member functions that modify/access elements
    // associated with a particular variable.
    mymap.insert({"Gilberto", 56});

    mymap["Beyonce"] = 41; // some types have operators

    // Operator [] is sometimes deemed unsafe for two reasons.
    // key doesn't exist: the code adds it to the map
    // key does exists: [] modifies it while insert doesn't
    fmt::print("(1) Rihanna is {0} years old\n", mymap["Rihanna"]);

    mymap.insert({"Rihanna", 35}); // nothing happens
    fmt::print("(2) Rihanna is {0} years old\n", mymap["Rihanna"]);

    mymap["Rihanna"] = 35;
    fmt::print("(3) Rihanna is {0} years old\n", mymap["Rihanna"]);

    return 0;
}
```

# C++ Maps

```
whovian@phy504:~/host$ g++ maps.cpp -o maps1 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps1
(1) Rihanna is 0 years old
(2) Rihanna is 0 years old
(3) Rihanna is 35 years old
```

Maps allow us to recover values from, for example, string keys

C++ has a special type for strings (we will cover them in detail later)

```
#include <map>
#include <string>
#include <fmt/core.h>

using std::map;    // better way to avoid using std::
using std::string; // don't pollute your code

int main ()
{
    // map<key type, value type> = initialize list
    // init the map with some key,values
    // We learned that we can use {} to init vars in C++
    map<string, int> mymap = {"Vivian", 37},
                      {"Marcelo", 38},
                      {"Vania", 58},
                      {"Isabella", 25};

    // Operator [] is sometimes deemed unsafe for two reasons.
    // key doesn't exist: the code adds it to the map
    // key does exists: [] modifies it while insert doesn't

    // .at() throws an exception if key does not exists.
    // Sometimes you want that strictness in your code, where you
    // are adamant to control when you want to insert new keys
    // (with .insert()) vs when you want to access known keys
    fmt::print("Rihanna is {}\n", mymap.at("Rihanna"));

    return 0;
}
```

# C++ Maps

Maps allow us to recover values from, for example, string keys

C++ offers different ways to access / modify elements

```
whovian@phy504:~/host$ g++ maps2.cpp -o maps2 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps2
terminate called after throwing an instance of 'std::out_of_range'
  what():  map::at
qemu: uncaught target signal 6 (Aborted) - core dumped
Aborted
```

```
#include <map>
#include <string>
#include <fmt/core.h>

using std::map; using std::string;

int main ()
{
    map<string, int> mymap = {{"Vivian", 37},
                               {"Marcelo", 38},
                               {"Vania", 58},
                               {"Beyonce", 41},
                               {"Rihanna", 35}};

    // How to get the number of keys?
    fmt::print("Number of elements: {}\n", mymap.size());

    // How to loop over map elements. May ways to do that
    for (auto it = mymap.begin(); it != mymap.end(); it++)
    { // iteration type is long (use auto!!!)
        fmt::print("Key: {}, Val: {}\n", it->first, it->second);
    }
    return 0;
}
```

# C++ Maps

Maps sort the keys

So, inserting new keys is expensive, but access is cheap

(opposite behavior:  
std::unordered\_map)

```
whovian@phy504:~/host$ g++ maps3.cpp -o maps3 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps3
Number of elements: 5
Key: Beyonce, Val: 41
Key: Marcelo, Val: 38
Key: Rihanna, Val: 35
Key: Vania, Val: 58
Key: Vivian, Val: 37
```

```

#include <map>
#include <string>
#include <fmt/core.h>

using std::map; using std::string;

int main ()
{
    map<string, int> mymap = {{"Vivian", 37}, {"Rihanna", 35};

    // C++ allows for map elements to be accessed via
    // Python style of loops in two different ways

    // First way: x is a {key, value} pair
    for (auto x : mymap)
    {
        // x is not a pointer. We access elements
        // first and second via "." and not "->"
        fmt::print("Key: {}, Val: {}\n", x.first, x.second);
    }

    fmt::print("\n");

    for (auto [key, val] : mymap)
    { // Second Way (C++17)
        fmt::print("Key: {}, Val: {}\n", key, val);
    }

    return 0;
}

```

```

whovian@phy504:~/host$ g++ maps4.cpp -o maps4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps4
Key: Rihanna, Val: 35
Key: Vivian, Val: 37
Key: Rihanna, Val: 35
Key: Vivian, Val: 37

```

# C++ Maps

C++ is compatible  
with Python style  
loops

Cool feature!

There is no better  
way. C++ provides  
many solutions to  
each problem!

```

#include <map>
#include <string>
#include <fmt/core.h>
#include <algorithm>

using std::map;
using std::string;
using std::for_each;

int main ()
{
    map<string, int> mymap = {{"Vivian", 37},
                               {"Rihanna", 35}};

    // Lambdas are anonymous functions in C++
    auto pf = [](auto x)
    {
        fmt::print("Key: {}, Val: {}\n", x.first, x.second);
    };
    // STL provides many useful algorithms
    for_each(mymap.begin(), mymap.end(), pf);

    fmt::print("\n");

    // you can also write a lambda func in-place (Python style)
    for_each(mymap.begin(), mymap.end(), [](auto x){
        fmt::print("Key: {}, Val: {}\n", x.first, x.second);
    });
}

return 0;
}

```

# C++ Maps

```

whovian@phy504:~/host$ g++ maps5.cpp -o maps5 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps5
Key: Rihanna, Val: 35
Key: Vivian, Val: 37
Key: Rihanna, Val: 35
Key: Vivian, Val: 37

```

Preview of STL  
algorithms and  
Lambda functions

Lambdas are  
insanely powerful

We will discuss  
them in detail later

# C++ Maps

```
#include <map>
#include <string>
#include <fmt/core.h>
#include <tuple>

using std::map;
using std::string;
using std::tuple;
using std::get;

int main ()
{
    map<string, tuple<int, string> > mymap =
        {{"Vivian", {37, "Miranda"}},
         {"Rihanna", {35, "Fenty"}}
    };

    for (auto [key, val] : mymap)
    {
        fmt::print("Key: {}, Age: {}, Surname: {}\n",
                  key, get<int>(val), get<string>(val));
    }

    return 0;
}
```

```
whovian@phy504:~/host$ g++ maps6.cpp -o maps6 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps6
Key: Rihanna, Age: 35, Surname: Fenty
Key: Vivian, Age: 37, Surname: Miranda
```

Maps can be combined with tuples to allow multiple values of different types

For multiple values of the same type, C++ offers [std::multimap](#)

```
#include <map>
#include <string>
#include <fmt/core.h>

using std::map; using std::string;

int main ()
{
    map<string, int> mymap = {{"Vivian", 37},
                               {"Rihanna", 35}};

    // Many ways to find if key exists
    auto x = "Beyonce";

    // C++20
    if (mymap.contains(x) == true) { // C++ bool
        fmt::print("key {0} exists\n", x);
    } else {
        fmt::print("key {0} does not exist\n", x);
    }

    if (mymap.count(x) > 0) {
        fmt::print("key {0} exists\n", x);
    } else {
        fmt::print("key {0} does not exist\n", x);
    }

    auto y = mymap.find(x);

    if (y != mymap.end()) {
        fmt::print("key {0} exists\n", x);
    } else {
        fmt::print("key {0} does not exist\n", x);
    }

    return 0;
}
```

# C++ Maps

```
whovian@phy504:~/host$ g++ maps7.cpp -o maps7 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps7
key Beyonce does not exist
key Beyonce does not exist
key Beyonce does not exist
```

How to find if a particular key exists in C++ maps?

Again, many options!

[.count\(\)](#) returns only 0 or 1  
(num of instances of a key)

[.find\(\)](#) returns {key, value} pair if it finds the searched key

```
#include <map>
#include <string>
#include <any>
#include <fmt/core.h>

using std::map;
using std::string;
using std::any;

int main ()
{
    map<string, any> mymap = {{"Vivian", string("Miranda")},
                               {"Rihanna", 35},
                               {"Beyonce", 500000000.0}};

    string x = std::any_cast<string>(mymap.at("Vivian"));
    fmt::print("Key: {0}, Val: {1} (surname)\n", "Vivian", x);

    int y = std::any_cast<int>(mymap.at("Rihanna"));
    fmt::print("Key: {0}, Val: {1} (age)\n", "Rihanna", y);

    double z = std::any_cast<double>(mymap.at("Beyonce"));
    fmt::print("Key: {0}, Val: {1} (fortune)\n", "Beyonce", z);

    return 0;
}
```

String literals are **const char\***, that cannot be casted to C++ strings. We need to create a C++ string from const char\* (using constructor)



# C++ Maps

What if we want to one (just one) value of different types

**std::any** (C++17)

```
whovian@phy504:~/host$ g++ maps8.cpp -o maps8 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps8
Key: Vivian, Val: Miranda (surname)
Key: Rihanna, Val: 35 (age)
Key: Beyonce, Val: 500000000 (fortune)
```

```

#include <map>
#include <string>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>

using std::map;
using std::string;
using boost::to_upper_copy;

int main ()
{
    // When I use maps I prefer to have keys all
    // uppercase. The Boost Library (informal
    // extension of STL) save us with two functions:
    // boost::to_upper_copy<std::string>
    // boost::to_upper

    map<string, int> mymap =
        {{to_upper_copy<string>("Vivian"), 37},
         {to_upper_copy<string>("Marcelo"), 38},
         {to_upper_copy<string>("Vania"), 58},
         {to_upper_copy<string>("Isabella"), 25},
         {to_upper_copy<string>("Rihanna"), 35}};

    fmt::print("{0} is {1} years old\n",
              to_upper_copy<string>("Rihanna"),
              mymap.at(to_upper_copy<string>("Rihanna")));

    return 0;
}

```

# C++ Maps

```

whovian@phy504:~/host$ g++ maps9.cpp -o maps9 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps9
RIHANNA is 35 years old

```

I prefer to have uppercase keys (avoid confusion)

STL has only a function to uppercase a single char

Boost Library comes to help us (super useful library)

```
#include <algorithm>
#include <fmt/core.h>
#include <list>

int main ()
{
    std::list<int> lt = {7, 5, 16, 8, 10};
    for (auto x : lt) // single line loop {} not needed
        fmt::print("{0}, ", x);
    fmt::print("\n");

    lt.push_back(13); // add to the back of the list
    lt.push_front(25); // add to the front of the list
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    // find elements in the list with the STL algorithm
    // remember that memory in linked lists is not contiguous
    auto it = std::find(lt.begin(), lt.end(), 16);
    if (it != lt.end())
        lt.insert(it, 42); // .insert() inserts value before pos
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    lt.push_front(13);
    lt.sort();
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    lt.unique(); // remove adjacent duplicates
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    return 0;
}
```

C++ has an implementation of **doubled linked lists** in the STL library

```
whovian@phy504:~/host$ g++ list.cpp -o list -std=c++20 -lfmt
whovian@phy504:~/host$ ./list
7, 5, 16, 8, 10,
25, 7, 5, 16, 8, 10, 13,
25, 7, 5, 42, 16, 8, 10, 13,
5, 7, 8, 10, 13, 13, 16, 25, 42,
5, 7, 8, 10, 13, 16, 25, 42,
```

```

#include <algorithm>
#include <fmt/core.h>
#include <list>

int main ()
{
    std::list<int> lt = {25, 7, 5, 16, 8, 10, 13};
    for (auto x : lt) // single line loop {} not needed
        fmt::print("{0}, ", x);
    fmt::print("\n");

    lt.pop_back(); // remove the last elem of the list
    lt.pop_front(); // remove the first elem of the list
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    // easy access to the first and last elem of the list
    fmt::print("{0}, {1}\n", lt.front(), lt.back());

    lt.reverse();
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    return 0;
}

```

C++ Lists

C++ has an implementation of **doubled linked lists** in the STL library

```

whovian@phy504:~/host$ g++ list2.cpp -o list2 -std=c++20 -lfmt
whovian@phy504:~/host$ ./list2
25, 7, 5, 16, 8, 10, 13,
7, 5, 16, 8, 10,
7, 10
10, 8, 16, 5, 7,

```

```

#include <fmt/core.h>
#include <list>
#include <algorithm>
using std::list; using fmt::print;
using std::unique_copy; using std::back_inserter;
using std::front_inserter; using std::inserter;

int main ()
{
    list<int> lt = {25, 7, 5, 25, 16, 10, 8, 10, 7, 6};
    lt.sort();
    for (auto x : lt)
        print("{0}, ", x);
    print("\n\n");

    // suppose we want to copy the list but without duplicates.
    list<int> lt2 = {100, 101, 102};
    unique_copy(lt.begin(), lt.end(), back_inserter(lt2));
    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");

    list<int> lt3 = {100, 101, 102};
    unique_copy(lt.begin(), lt.end(), front_inserter(lt3));
    for (auto x : lt3)
        print("{0}, ", x);
    print("\n\n");

    list<int> lt4 = {100, 101, 102};
    unique_copy(lt.begin(), lt.end(), inserter(lt4, lt4.begin()));
    for (auto x : lt4)           inserter(list, pos)
        print("{0}, ", x);      inserts before pos
    print("\n\n");

    list<int> lt5 = {100, 101, 102, 103, 104};
    auto x = std::find(lt5.begin(), lt5.end(), 102);
    unique_copy(lt.begin(), lt.end(), inserter(lt5, x));
    for (auto x : lt5)
        print("{0}, ", x);
    print("\n\n");
}

```

# C++ Lists

STL algorithm offers powerful abstract tools to manipulate **doubled linked lists**

**inserters** saved us from memory corruption (lt2, lt3, lt4, lt5 not large enough)

```

whovian@phy504:~/host$ g++ list3.cpp -o list3 -std=c++20 -lfmt
whovian@phy504:~/host$ ./list3
5, 6, 7, 7, 8, 10, 10, 16, 25, 25,
100, 101, 102, 5, 6, 7, 8, 10, 16, 25,
25, 16, 10, 8, 7, 6, 5, 100, 101, 102,
5, 6, 7, 8, 10, 16, 25, 100, 101, 102,
100, 101, 5, 6, 7, 8, 10, 16, 25, 102, 103, 104,

```

```

#include <fmt/core.h>
#include <list>
#include <algorithm>

using std::list;
using fmt::print;
using std::reverse_copy;
using std::back_inserter;
using std::inserter;

int main ()
{
    list<int> lt = {25, 7, 5, 25, 16, 10, 8, 10, 7, 6};
    lt.sort();
    lt.unique();
    for (auto x : lt)
        print("{0}, ", x);
    print("\n\n");

    list<int> lt2 = {};
    reverse_copy(lt.begin(), lt.end(), back_inserter(lt2));
    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");

    list<int> lt3 = {100, 101, 102, 103, 104};
    auto x = std::find(lt3.begin(), lt3.end(), 102);
    reverse_copy(lt.begin(), lt.end(), inserter(lt3, x));
    for (auto x : lt3)
        print("{0}, ", x);
    print("\n\n");
}

```

```

whovian@phy504:~/host$ g++ list5.cpp -o list5 -std=c++20 -lfmt
whovian@phy504:~/host$ ./list5
5, 6, 7, 8, 10, 16, 25,
25, 16, 10, 8, 7, 6, 5,
100, 101, 25, 16, 10, 8, 7, 6, 5, 102, 103, 104,

```

# C++ Lists

Algorithms don't have permission to delete elements

STL algorithm offers powerful abstract tools to manipulate **doubled linked lists**

**ininserters** saved us from memory corruption

```

#include <fmt/core.h>
#include <list>
#include <algorithm>

using std::list;
using fmt::print;
using std::unique_copy;

int main ()
{
    list<int> lt = {25, 07, 05, 25, 16, 10, 8, 10, 07, 06};
    list<int> lt2 = {01, 01, 01, 01, 01, 01, 1, 01, 01, 01};

    lt.sort();
    for (auto x : lt)
        print("{0}, ", x);
    print("\n\n");

    // no need for inserters (lt2 has enough capacity)
    unique_copy(lt.begin(), lt.end(), lt2.begin());
    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");

    return 0;
}

```

```

whovian@phy504:~/host$ g++ list4.cpp -o list4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./list4
5, 6, 7, 7, 8, 10, 10, 16, 25, 25,
5, 6, 7, 8, 10, 16, 25, 1, 1, 1,

```

# C++ Lists

STL algorithm offers powerful abstract tools to manipulate **doubled linked lists**

**inserters** saved us from memory corruption (lt2, lt3, lt4, lt5 not large enough)

Algorithms don't have permission to delete elements

```
#include <fmt/core.h>
#include <list>
#include <algorithm>

using std::list; using fmt::print; using std::unique_copy;

int main ()
{
    list<int> lt = {25, 07, 05, 25, 16, 10, 8, 10, 07, 06};
    list<int> lt2 = {01, 01, 01, 01, 01, 01, 1, 01, 01, 01};
    lt.sort();
    lt.unique();
    for (auto x : lt)
        print("{0}, ", x);
    print("\n\n");

    auto ptr = unique_copy(lt.begin(), lt.end(), lt2.begin());
    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");

    // now we will show that ptr iterator points
    // to the last element of the unique_copy
    for (auto x = lt2.begin(); x != ptr; x++) {
        print("{0}, ", *x); // x is an iterator (restricted version
                            //                      of a pointer)
    }
    print("\n\n");

    // How to permanently erase the last elements?
    lt2.erase(ptr, lt2.end());

    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");
}
```

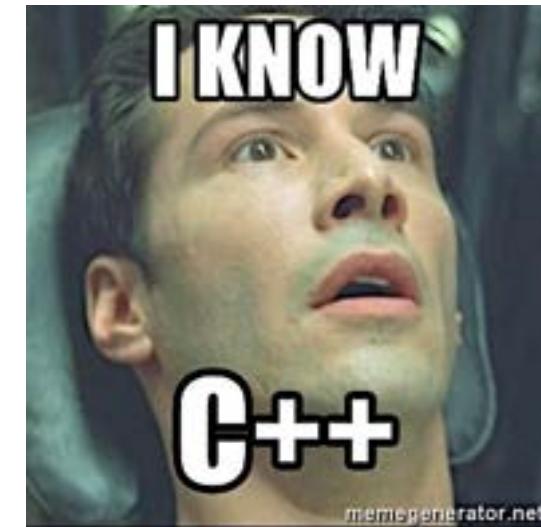
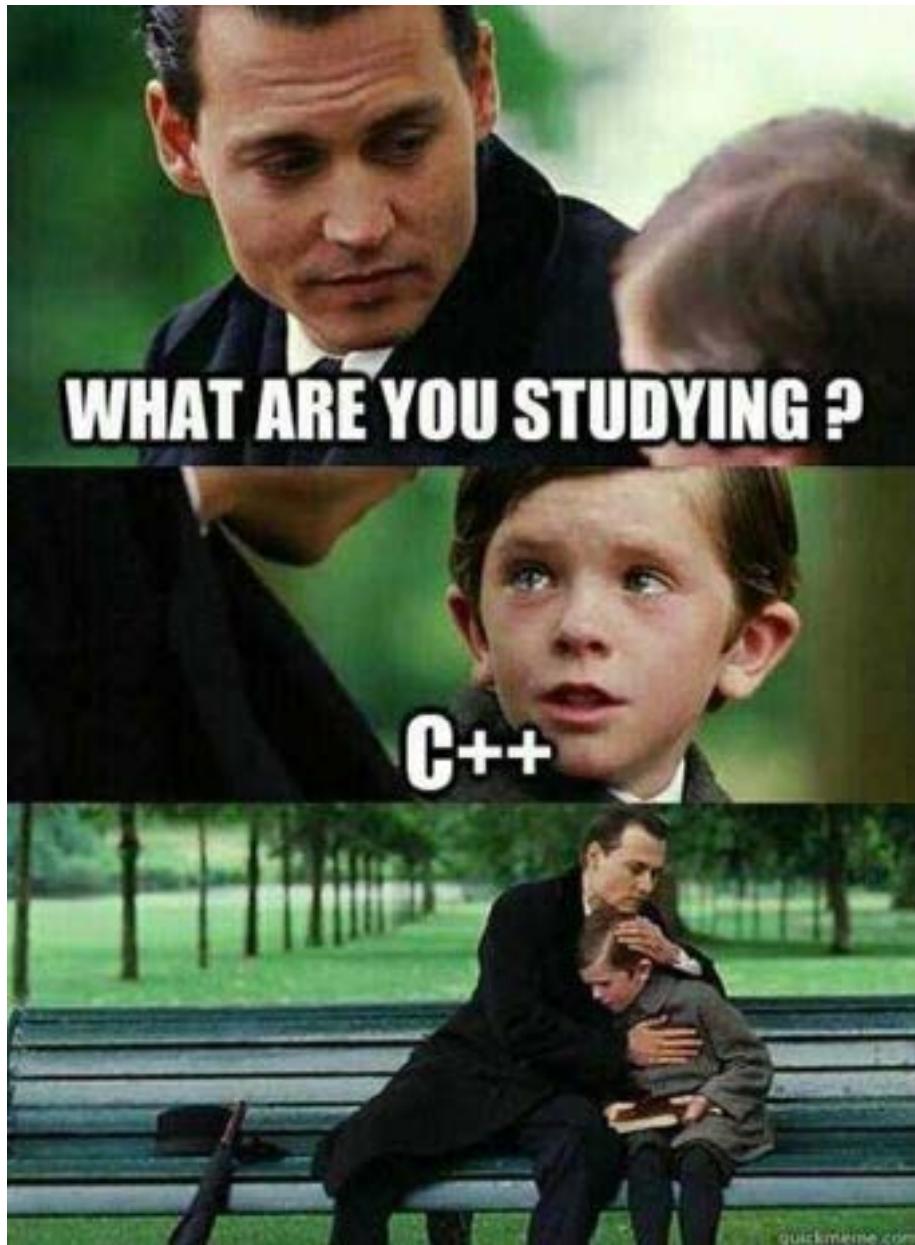
# C++ Lists

**Algorithms** don't have permission to delete elements

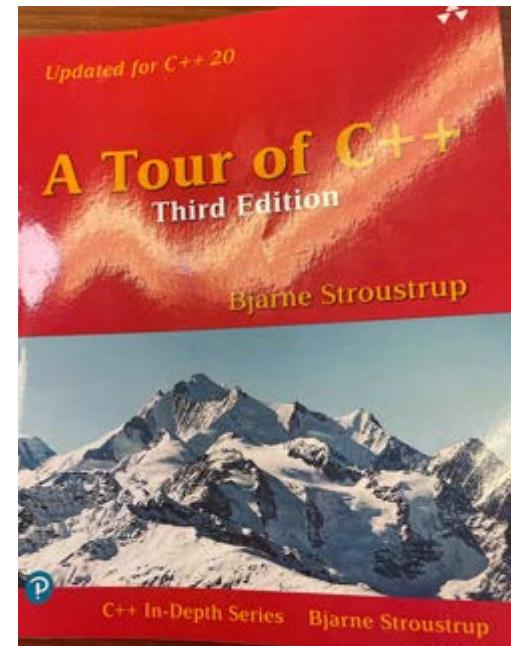
But they do return position where they stop

```
whovian@phy504:~/host$ g++ list6.cpp -o list6 -std=c++20 -lfmt
whovian@phy504:~/host$ ./list6
5, 6, 7, 8, 10, 16, 25,
5, 6, 7, 8, 10, 16, 25, 1, 1, 1,
5, 6, 7, 8, 10, 16, 25,
5, 6, 7, 8, 10, 16, 25,
```

# Lecture 28: C++ part IV



## Suggested Literature



```
#include <string>
#include <fmt/core.h>
// adapted from
// geeksforgeeks.org/c-string-class-and-its-applications/
using std::string; using fmt::print;

int main ()
{
    string str1("Vivian Miranda"); // init by string literal
    string str2 = {"Vivian Miranda"}; // init by string literal
    string str3{"Vivian Miranda"}; // init by string literal

    string str4(str1); // init by another string

    string str5(5, '#'); // init by char with number of occurrence

    // init by part of another string
    string str6(str1, 5, 5); // from 5th index (second parameter)
                           // 5 characters (third parameter)

    // init by part of another string : iterator version
    string str7(str1.begin(), str1.begin() + 5);

    print("{0}, {1}, {2}, {3} \n", str1, str2, str3, str4);
    print("{0}\n", str5);
    print("{0}\n", str6);
    print("{0}\n", str7);

    return 0;
}
```

# C++ Strings

C++ offers more powerful string manipulation compared to C (either via **STL** or **Boost Library**)

```
whovian@phy504:~/host$ g++ strings.cpp -o string1 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string1
Vivian Miranda, Vivian Miranda, Vivian Miranda, Vivian Miranda
#####
n Mir
Vivia
```

```

#include <string>
#include <fmt/core.h>
// adapted from
// geeksforgeeks.org/c-string-class-and-its-applications/
using std::string; using fmt::print;

int main ()
{
    string str1("Vivian Miranda");      // init by string literal

    string str2 = str1; // (assignment)

    const int len = str2.length();
    print("Length of str1 string is: {0}\n", len);

    print("str2: {0} \n", str2);
    str2.clear(); // deletes all character from string
    print("str2: {0} \n", str2);

    char ch1 = str1.at(2); // safer version as throws out_of_range
                          // exception if arg is out of range
    char ch2 = str1[3];
    print("ch1: {0}; ch2: {1} \n", ch1, ch2);

    char ch3 = str1.front(); // Same as str1[0]
    char ch4 = str1.back(); // Same as str1[str1.length()-1]
    print("ch3: {0}; ch4: {1} \n", ch3, ch4);

    return 0;
}

```

C++ Strings

C++ difference between **.at()** member function and **operator[]** is always safety. The former checks if out of bounds and throws an exception if not

```

whovian@phy504:~/host$ g++ strings2.cpp -o string2 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string2
Length of str1 string is: 14
str2: Vivian Miranda
str2:
ch1: v; ch2: i
ch3: V; ch4: a

```

```
#include <string>
#include <fmt/core.h>
// adapted from
// geeksforgeeks.org/c-string-class-and-its-applications/
using std::string; using fmt::print;

int main () {
    string str1("Vivian Miranda"); // init by string literal

    // .c_str() returns null terminated char array the C++ string
    // holds under the hood. The name .c_str() is short of C strings.
    // .c_str() helps compatibility with C libraries

    // .c_str() is super dangerous. If someone cast from const char*
    // to char*, the code can corrupt the vars the C++ string holds
    const char* charstr = str1.c_str();
    printf("str1: \"%s\"\n", charstr);

    str1.append(", Ph.D."); // add the arg string at the end
    print("str1: \"{0}\"\n", str1);

    string str2 = "Mira";
    if (str1.find(str2) != string::npos)
    { // x.find() = find if substring arg is in inside string x
        print("Found str2: \"{0}\" inside str1: \"{1}\"\n", str2, str1);
    }

    // more on substrings
    print(
        "Substring: \"{0}\" (from 7th char, len=3) of string str1: \"{1}\"\n",
        str1.substr(7, 3), str1);

    print("Substring \"{0}\" (from 7th char) of string str1: \"{1}\"\n",
        str1.substr(7), str1);
}

return 0;
}
```

# C++ Strings

C++ string **.c\_str()**  
member function  
provides users the  
possibility of using  
C libraries that read  
(but not change!)  
C-style **const**  
**char\*** null-  
terminated strings

```
whovian@phy504:~/host$ ./string3
str1: "Vivian Miranda"
str1: "Vivian Miranda, Ph.D."
Found str2: "Mira" inside str1: "Vivian Miranda, Ph.D."
Substring: "Mir" (from 7th char, len=3) of string str1: "Vivian Miranda, Ph.D."
Substring "Miranda, Ph.D." (from 7th char) of string str1: "Vivian Miranda, Ph.D."
```

```
#include <string>
#include <fmt/core.h>
// adapted from
// geeksforgeeks.org/c-string-class-and-its-applications/
using std::string; using fmt::print;

int main () {
    string str1("Vivian Miranda"); // init by string literal
    string str2 = str1;
    print("str1: \"{}\"\n", str1);

    str2.erase(7, 4); // erase from 7th char, 4 chars
    print("str2: \"{}\"\n", str2);

    // erase iterator version
    string str3 = str1;
    str3.erase(str3.begin() + 3, str3.end() - 3);
    print("str3: \"{}\"\n", str3);

    string str4 = str1;
    str4.replace(0, 6, "ViV   ");
    print("str4: \"{}\"\n", str4);

    return 0;
}
```

# C++ Strings

There are many C++ string **STL** member functions (**.something()**) that alter / change the content of a string

```
whovian@phy504:~/host$ g++ strings4.cpp -o string4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string4
str1: "Vivian Miranda"
str2: "Vivian nda"
str3: "Vivnda"
str4: "ViV   Miranda"
```

```

#include <string>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>

using std::string;
using fmt::print;
using boost::to_upper;
using boost::to_upper_copy;

int main ()
{
    string str1("Vivian Miranda"); // init by string literal
    print("str1: \"{}\"\n", str1);

    string str2 = str1;
    for (auto& x : str2) // STL way to uppercase strings (ugly!)
    { // auto& = pass by reference (new concept!)
        x = toupper(x);
    }
    print("str2: \"{}\"\n", str2);                                & = lvalue reference
                                                                (lecture10)

    // BOOST Library offers much better functionality
    string str3 = str1;
    boost::to_upper(str3);
    print("str3: \"{}\"\n", str3);

    // via copy (not affecting the original string)
    string str4 = to_upper_copy(str1);
    print("str4: \"{}\"\n", str4);

    // below, we need to type the template parameter explicitly
    // because it cannot be inferred from the string literal
    // string literal != std::string
    string str5 = to_upper_copy<string>("Vivian Miranda");
    print("str5: \"{}\"\n", str5);

    return 0;
}

```

```

whovian@phy504:~/host$ g++ strings5.cpp -o string5 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string5
str1: "Vivian Miranda"
str2: "VIVIAN MIRANDA"
str3: "VIVIAN MIRANDA"
str4: "VIVIAN MIRANDA"
str5: "VIVIAN MIRANDA"

```

# C++ Strings

## Boost Library

functions are a bit more complete.

Example: uppercase or lowercase the entire string instead of doing **char** by **char** (STL)

Issue w/ boost. Install not trivial/portable

# Boost Library

Testing ground of features to be added in new standards  
(some libs will never be added but are still useful)

The screenshot shows the Boost 1.82.0 Library Documentation homepage. At the top, there's a green banner with the Boost logo and a quote from Herb Sutter and Andrei Alexandrescu about C++ Coding Standards. Below the banner, the page title is "BOOST 1.82.0 LIBRARY DOCUMENTATION". There are search and filter options, including "All Categorized Condensed" and "Sort by: Name First Release C++ Minimum". A large red button labeled "GET BOOST" is prominently displayed. On the right, there's a sidebar with links for "ENHANCED BY Go", a search bar, a "Donate" button, and a "WELCOME" section with links to "INTRODUCTION", "COMMUNITY", "DEVELOPMENT", and "DOCUMENTATION". The main content area lists several library components with their details:

- Accumulators**: Framework for incremental calculation, and collection of statistical accumulators.
  - Author(s): Eric Niebler
  - First Release: 1.36.0
  - C++ Standard Minimum Level: 03
  - Categories: Math and numerics
- Algorithm**: A collection of useful generic algorithms.
  - Author(s): Marshall Clow
  - First Release: 1.50.0
  - C++ Standard Minimum Level: 03
  - Categories: Algorithms
- Align**: Memory alignment functions, allocators, traits.
  - Author(s): Glen Fernandes
  - First Release: 1.56.0
  - C++ Standard Minimum Level: 03
  - Categories: Memory
- Any**: Safe, generic container for single values of different value types.
  - Author(s): Kevin Henney
  - First Release: 1.23.0
  - C++ Standard Minimum Level: 03
  - Categories: Data structures
- Array**: STL compliant container wrapper for arrays of constant size.
  - Author(s): Nicolai Josuttis
  - First Release: 1.17.0
  - C++ Standard Minimum Level: 03
  - Categories: Containers

A vertical sidebar on the right lists "Getting Started Libraries" with their versions: 1.82.0 (Current Release), 1.81.0, 1.80.0, 1.79.0, 1.78.0, 1.77.0, 1.76.0, 1.75.0, 1.74.0, 1.73.0, 1.72.0, 1.71.0, 1.70.0, 1.69.0, 1.68.0, 1.67.0, 1.66.0, 1.65.1, 1.65.0, 1.64.0, 1.63.0, 1.62.0.

```

#include <string>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
#include <boost/lexical_cast.hpp>

using std::string; using fmt::print;
using boost::lexical_cast; using std::to_string;

int main ()
{
    string str1("1.23456"); // init by string literal
    double x1 = lexical_cast<double>(str1); // if fails, then the
                                              // code doesn't compile!
    double x2 = stod(str1);
    print("x1 = {0}; x2 = {1} \n", x1, x2);

    // We should avoid strings with multiple numbers (we will learn soon)
    string str2 = "1.23456 2.34567";
    std::string::size_type sz; // size_type is basically unsigned int
    double x3 = stod(str2, &sz); // second argument useful when the
                               // string contains multiple numbers
    double x4 = stod(str2.substr(sz)); // .substr(sz) = create substring
    print("x2 = {0}; x3 = {1} \n", x3, x4);

    string str3("3"); // init by string literal
    int x5 = lexical_cast<int>(str3);
    int x6 = stoi(str3);
    print("x5 = {0}; x6 = {1} \n", x5, x6);

    double num = 3.1415;
    string str4 = to_string(num);  to_string is part of the STL
    string str5 = lexical_cast<string>(num); // if fails, then the
                                              // code doesn't compile!
    print("str4 = {0}; str5 = {1} \n", str4, str5);

    return 0;
}

```

# C++ Strings

```

whovian@phy504:~/host$ g++ strings6.cpp -o string6 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string6
x1 = 1.23456; x2 = 1.23456
x2 = 1.23456; x3 = 2.34567
x5 = 3; x6 = 3
str4 = 3.141500; str5 = 3.1415000000000002

```

**STL** offers functions to cast strings to numbers and vice-versa.

I prefer [boost::lexical cast](#) because it provides [unified API](#)

```
#include <string>
#include <fmt/core.h>

using std::string;
using fmt::print;

int main ()
{
    string str1("vivian");
    string str2("vivian");
    if (str1 == str2)
        print("str1 and str2 are equal\n");

    string str3 = "miranda";
    if (str1 > str3)
        print("str1 > str3 lexicographically\n");
    else if (str1 < str3)
        print("str1 < str3 lexicographically\n");
    else
        print("str1 and str3 are equal\n");

    // you can also use .compare() - this function
    // has extra arguments that allows the user to
    // compare substrings
    auto x = str3.compare(str1); // only sign matters
    auto y = str1.compare(str3); // only sign matters
    print("x = {0}, y = {1}\n", x, y);

    return 0;
}
```

```
whovian@phy504:~/host$ g++ strings7.cpp -o string7 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string7
str1 and str2 are equal
str1 > str3 lexicographically
x = -1, y = 6
```

# C++ Strings

Locale insensitive but  
case sensitive  
comparisons.

If you do need locale  
sensitive comparison,  
the C **STL** function  
**strcoll** is simple and  
effective

```
#include <string>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
using std::string; using fmt::print;
using namespace boost::algorithm;
// code adapted from
// geeksforgeeks.org/boosttrim-in-cpp-library/
// geeksforgeeks.org/c-boost-string-algorithms-library/
int main () {
    string s1 = "VivianMirandaPhy504";
    string s2 = "VivianMirandaPhy504";
    string s3 = "VivianMirandaPhy504";
    trim_left(s1);
    trim_right(s2);
    trim_left(s3);

    print("trimmed left s1: |{0}|\n", s1);
    print("trimmed right s2: |{0}|\n", s2);
    print("trimmed both sides s2: |{0}|\n\n", s3);

    string s4 = "viv1viv2viv3viv4viv5viv6";
    print("first_copy: {0}\n", erase_first_copy(s4, "viv"));
    print("nth_copy: {0}\n", erase_nth_copy(s4, "viv", 3));
    print("last_copy: {0}\n", erase_last_copy(s4, "viv"));
    print("all_copy: {0}\n\n", erase_all_copy(s4, "viv"));

    string s5 = "Vivian Miranda";
    print("erase_head_copy: {0}\n", erase_head_copy(s5,7));
    print("erase_tail_copy: {0}\n\n", erase_tail_copy(s5,7));

    string s6 = "Vivian-Miranda-1986-37";
    print("first_copy: {0}\n", replace_first_copy(s6,"-","_"));
    print("last_copy: {0}\n", replace_last_copy(s6,"-","_"));
    print("all_copy: {0}\n", replace_all_copy(s6,"-","_"));
}
```

# C++ Strings

Boost Library offers  
good string  
functionality  
boost/algorithm/string

```
trimmed left s1: |VivianMirandaPhy504|
trimmed right s2: |VivianMirandaPhy504|
trimmed both sides s2: |VivianMirandaPhy504|
first_copy: 1viv2viv3viv4viv5viv6
nth_copy: viv1viv2viv34viv5viv6
last_copy: viv1viv2viv3viv4viv56
all_copy: 123456

erase_head_copy: Miranda
erase_tail_copy: Vivian

first_copy: Vivian_Miranda-1986-37
last_copy: Vivian-Miranda-1986_37
all_copy: Vivian_Miranda_1986_37
```

```

#include <string>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
#include <boost/lexical_cast.hpp>
using std::string; using fmt::print;
using namespace boost::algorithm;
using boost::lexical_cast; using boost::iequals;
C++ has a proper bool type
bool convert_string (const std::string s) {
    if (iequals(s, string("T")) || 
        iequals(s, string("True")) || 
        iequals(s, string("Yes")))
    {
        return true;
    }
    else if (iequals(s, std::string("F")) || 
              iequals(s, std::string("False")) || 
              iequals(s, std::string("No")))
    {
        return false;
    }
    else
    {
        const double tmp = lexical_cast<double>(s);
        return ((tmp < 0) == false) ? true : false;
    }
}
int main()
{
    bool x1 = convert_string("True");
    bool x2 = convert_string("truE");
    bool y1 = convert_string("False");
    bool y2 = convert_string("FaLSe");
    print("{0}, {1}, {2}, {3}\n", x1, x2, y1, y2);
}

```

# C++ Strings

General conversion to **bool**  
via string comparison.

Difference between  
**boost::iequals** and STL  
**string.compare** (or **==**)?

**boost::iequals** is case  
insensitive!

```

whovian@phy504:~/host/CPP$ g++ strings9.cpp -o string9 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string9
true, true, false, false

```

```

#include <string>
#include <vector>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
#include <boost/lexical_cast.hpp>
using std::string; using fmt::print;
using namespace boost::algorithm;
using std::vector; using boost::split;

int main() {
    std::string content =
        "First line Content\n"
        "Second line Content\n"
        "Third Line Content";

    vector<string> lines;
    boost::split( lines, content,
                 boost::is_any_of("\n"),
                 boost::token_compress_on
               );
    for (auto x : lines) {
        print("{0}\n", x);
    }
    print("\n");

    for(auto line : lines) {
        vector<string> words;

        boost::split( words, line,
                     boost::is_any_of(" \t"),
                     boost::token_compress_on
                   );
        for (auto x : words) {
            print("{0}\n", x);
        }
        print("\n");
    }
}

```

# C++ Strings

**Boost Library** offers easy interface to split text into lines and lines into words

## boost/algorithm/string

```

whovian@phy504:~/host/CPP$ g++ strings10.cpp -o string10 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string10
First line Content
Second line Content
Third Line Content

First
line
Content

Second
line
Content

Third
Line
Content

```

```

#include <string>
#include <vector>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
#include <boost/lexical_cast.hpp>
using std::string; using fmt::print;
using namespace boost::algorithm;
using std::vector; using boost::split;
using boost::starts_with; using boost::is_any_of;

int main()
{
    string content = "#First line Content";
    vector<string> words;
    split( words,
           content,
           is_any_of(" \t"),
           boost::token_compress_on
    );
    if(starts_with(words[0], "#"))
    {
        print("This is a comment line\n"); key
                                         value
    }

    // DEAL WITH key=value (no space) line
    vector<string> words2 = {"key=value"};
    auto pos = words2.at(0).find("=");
    if (pos != std::string::npos)
    {
        string key = words2.at(0);
        words2.resize(0); // zero size (reset vector)
        split( words2, key,
               boost::is_any_of("-"),
               boost::token_compress_on
        );
        for (auto x : words2) {
            print("{0}\n", x);
        }
    }
}

```

# C++ Strings

## Boost Library offers good string functionality

### boost/algorithm/string

```

whovian@phy504:~/host/CPP$ g++ strings11.cpp -o string11 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string11
This is a comment line

```

We will be able to check if the line is a comment (starts with **#**), and to separate **key=value** lines (no spaces)

```
#include <string>
#include <fmt/core.h>
#include <boost/regex.hpp>
#include <boost/algorithm/string/regex.hpp>
using std::string; using fmt::print;
using boost::regex; using boost::regex_match;

int main()
{
    regex re_is_equal("^[=]+$");
    regex re_is_comma("^[,]+$");
    regex re_is_semicolon("^[;]+$");

    string s1 = "=";
    if (regex_match(s1, re_is_equal)) {
        print("s1 matches re_is_equal\n");
    }
    string s2 = ",,,";
    if (regex_match(s2, re_is_comma)) {
        print("s2 matches re_is_comma\n");
    }
    string s3 = ";";
    if (regex_match(s3, re_is_semicolon)) {
        print("s3 matches re_is_semicolon\n");
    }
    string s4 = ";;;";
    if (regex_match(s4, re_is_semicolon)) {
        print("s4 matches re_is_semicolon\n");
    }
}
```

# C++ Strings

Boost Library and STL offer support for regular expressions

```
whovian@phy504:~/host/CPP$ nano strings12.cpp
whovian@phy504:~/host/CPP$ g++ strings12.cpp -o string12 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string12
s1 matches re_is_equal
s2 matches re_is_comma
s3 matches re_is_semicolon
s4 matches re_is_semicolon
```

## std::regex\_match

(warning: a few regex commands may require double \\ - example \\d for digits)

```

#include <string>
#include <fmt/core.h>
#include <boost/regex.hpp>
#include <boost/algorithm/string/regex.hpp>
using std::string; using fmt::print;
using boost::regex; using boost::regex_match;

int main()
{
    regex re_has_equal("^(A-Za-z)+?=[A-Za-z]+)?$");

    string s1 = "viv=";
    if (regex_match(s1, re_has_equal)) {
        print("s1 matches re_has_equal\n");
    }
    string s2 = "viv=viv";
    if (regex_match(s2, re_has_equal)) {
        print("s2 matches re_has_equal\n");
    }
    string s3 = "=viv";
    if (regex_match(s3, re_has_equal)) {
        print("s3 matches re_has_equal\n");
    }
    string s4 = "==viv";
    if (regex_match(s4, re_has_equal)) {
        print("s4 matches re_has_equal\n");
    }
    string s5 = "ViV=viv";
    if (regex_match(s5, re_has_equal)) {
        print("s5 matches re_has_equal\n");
    }
    string s6 = "ViVviv";
    if (!regex_match(s6, re_has_equal)) {
        print("s6 doesn't match re_has_equal\n");
    }
}

```

# C++ Strings

## Boost Library and STL offer support for regular expressions

```

whovian@phy504:~/host/CPP$ g++ strings14.cpp -o string14 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string14
s1 matches re_has_equal
s2 matches re_has_equal
s3 matches re_has_equal
s4 matches re_has_equal
s5 matches re_has_equal
s6 doesn't match re_has_equal

```

## std::regex\_match

(warning: a few regex commands may require double \\ - example \\d for digits)

```

#include <string>
#include <fmt/core.h>
#include <boost/regex.hpp>
#include <boost/algorithm/string/regex.hpp>
using std::string; using fmt::print;
using boost::regex; using boost::regex_match;

int main()
{
    // doesn't match scientific notation ex: 1e+20
    regex re_is_float("[+-]?\\d+.*\\d*");
    string s1 = "7.";
    if (regex_match(s1, re_is_float)) {
        print("s1 matches re_is_float\n");
    }
    string s2 = "17.32";
    if (regex_match(s2, re_is_float)) {
        print("s2 matches re_is_float\n");
    }

    regex re_is_number("[+-]?\\d+.*\\d*[eE][+-]?\\d*");
    string s3 = "17.32";
    if (regex_match(s3, re_is_number)) {
        print("s3 matches re_is_number\n");
    }
    string s4 = "1.32e+20";
    if (regex_match(s4, re_is_number)) {
        print("s4 matches re_is_number\n");
    }
}

```

C++ Strings  
**Boost Library** and  
**STL** offer support for  
regular expressions

need double \\

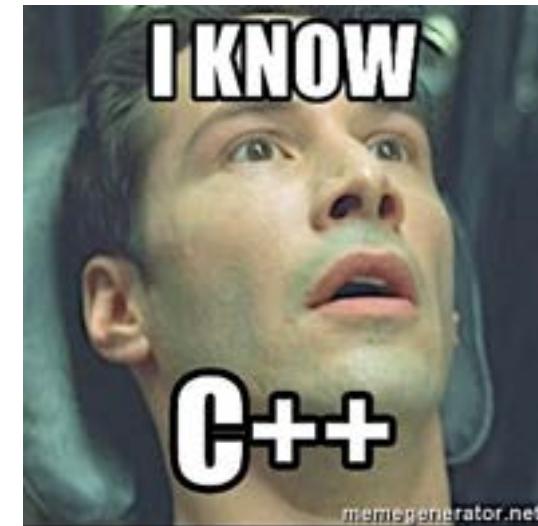
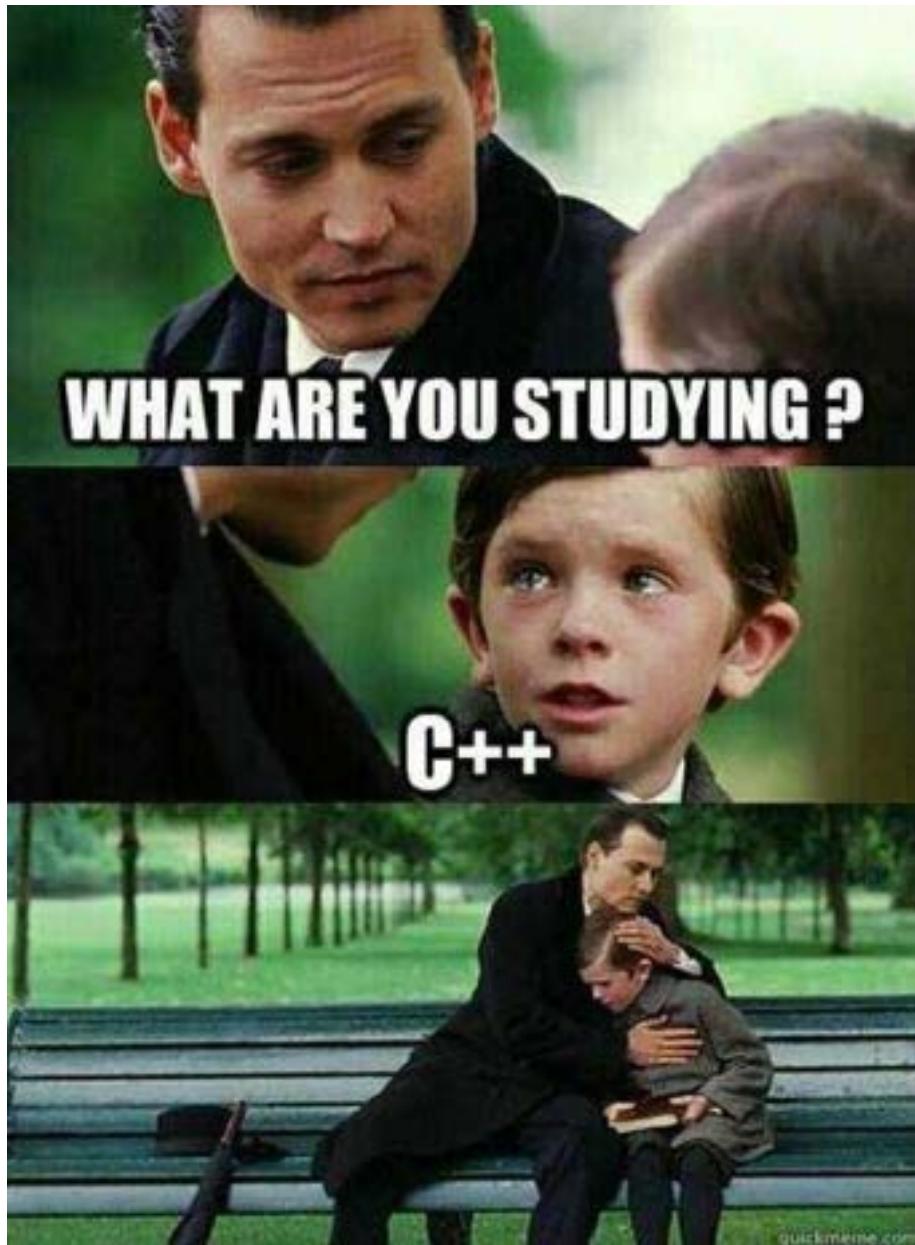
```

whovian@phy504:~/host/CPP$ nano strings13.cpp
whovian@phy504:~/host/CPP$ g++ strings13.cpp -o string13 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string13
s1 matches re_is_float
s2 matches re_is_float
s3 matches re_is_number
s4 matches re_is_number

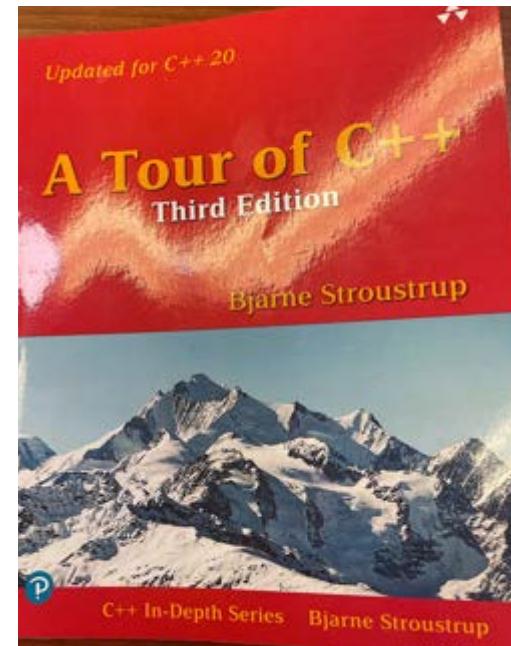
```

need double \\  
because \\d \\t \\n has a  
special meaning

# Lecture 29: C++ part V



## Suggested Literature



```

#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip> // setw, setfill, setprecision
#include <fmt/print.h>
using std::cout; using std::endl;
using fmt::print; using std::setfill;
using std::setprecision; using std::setw;

int main ()
{ // Before C++23, IO was dealt with streams
    const double x = 7.1234567890;
    const int y = 730;

    cout << "x = " << setprecision(6) << x << endl;
    cout << "y = " << setw(5) << y << endl;
    cout << "y = " << setfill('0') << setw(5) << y << endl;
    cout << "y = " << setfill('*') << setw(5) << y << endl;
    cout << endl;

    // fmt offers a preview of C++23
    print("x = {:.5f}\n", x); // = setprecision(6)
    print("y = {:5d}\n", y);
    print("y = {:05d}\n", y);
    // > is an alignment specifier
    // fmt.dev/latest/syntax.html#format-specification-mini-language
    print("y = {:{*>}5d}\n\n", y);

    // fmt offers a type safer version of printf
    // If format fails, it throws fmt::v9::format_error
    fmt::printf("x = %.5f\n", x);
    fmt::printf("y = %5d\n", y);
}

```

C++ IO  
Old C++ adopts **streams** for both input and output

C++23 add the command **print()** that has Python style API for formatting

```

whovian@phy504:~/host/CPP$ g++ io1.c -o io1 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io1
x = 7.12346
y = 730
y = 00730
y = **730

x = 7.12346
y = 730
y = 00730
y = **730

x = 7.12346
y = 730

```

```

#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip>

using std::cout; using std::endl;
using fmt::print; using std::setfill;
using std::boolalpha;

int main ()
{
    const bool a = false;
    const bool b = true;
    cout << "a = " << a << " b = " << b << endl;
    cout << boolalpha;
    cout << "a = " << a << " b = " << b << endl;

    print("a = {}, b = {}\n", a, b);

    // print alignment
    const int y = 730;
    print("{:>20}\n", y); // left
    print("{:>20}\n", y); // center
    print("{:<20}\n", y); // right
}

```

# C++ IO

Printing **bool** with **streams** requires an extra flag, otherwise **streams** prints an integer

Alignment specifier is a cool feature of **fmt::print()**

```

whovian@phy504:~/host/CPP$ g++ io2.c -o io2 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io2
a = 0 b = 1
a = false b = true
a = false, b = true
*****730*****
*****730*****
730*****

```

```
#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip>
using fmt::print;
using namespace std;

int main ()
{
    const int y = 125;
    cout << left << "|" << setfill('*') <<
        setw(10) << y << "|" << endl;
    cout << right << "|" << setfill('*') <<
        setw(10) << y << "|" << endl;
    // no center position!
    cout << endl;

    const double x = 300545900;
    cout << "x = " << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        defaultfloat << x << endl;

    cout << "x = " << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        scientific << x << endl;

    cout << "x = " << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << x << endl;
    cout << endl;

    // print offers more compact notation
    print("x = {::*>21.10g}\n", x);
    print("x = {::*>21.10e}\n", x);
    print("x = {::*>21.10f}\n", x);
    return 0;
}
```

# C++ IO

**streams** allows the control of alignment (but not centering)

Scientific notation in **streams**  
vs **fmt::print()**

Print has more compact  
notation than streams

```
whovian@phy504:~/host/CPP$ g++ io3.c -o io3 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io3
|125*****|
*****125

x = *****300545900
x = *****3.0054590000e+08
x = *300545900.0000000000

x = *****300545900
x = *****3.0054590000e+08
x = *300545900.0000000000
```

```

#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip>
using fmt::print;
using namespace std;

int main ()
{
    const double x = 3125;
    cout << "x = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        noshowpoint << x << endl;

    cout << "x = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        showpoint << x << endl;
    cout << endl;

    const double y = 1.2;
    cout << "y = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        noshowpoint << y << endl;

    cout << "y = " << right << setfill('*') <<
        setw(15) << showpoint << y << endl;
    cout << endl;

    return 0;
}

```

# C++ IO

```

whovian@phy504:~/host/CPP$ g++ io4.c -o io4 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io4
x = ****3125
x = *****3125.000
y = *****1.2
y = *****1.200000

```

**streams** additional

manipulators:

**std::showpoint /**  
**std::noshowpoint**

**std::noshowpoint** hides .  
on integers (3125 vs  
3125.000)

```

#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip>
using fmt::print;
using namespace std;

int main ()
{
    const double x = 3125;
    cout << "x = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        scientific << uppercase << x << endl;

    cout << "x = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        scientific << nouppercase << x << endl;
    cout << endl;

    const int y = 37;
    cout << "y = " << right << setfill('*') <<
        setw(15) << fixed << showbase <<
        dec << y << endl;

    cout << "y = " << right << setfill('*') <<
        setw(15) << fixed << showbase <<
        hex << y << endl;
    cout << endl;

    return 0;
}

```

# C++ IO

```

whovian@phy504:~/host/CPP$ g++ io5.c -o io5 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io5
x = **3.1250000E+03
x = **3.1250000e+03

y = ****37
y = *****0x25

```

**streams** additional  
manipulators: **std::  
uppercase / std::  
nouppercase;**  
**std::showbase / std::  
noshowbase** and **std::hex  
/ std::dec**

```

#include <iostream>
#include <fmt/core.h>
#include <vector>
#include <fmt/ranges.h> // fmt::join
using fmt::print; using fmt::join; using std::endl;
using std::vector; using std::cout;

int main ()
{
    vector<double> v = {1.0, 2.0, 3.0, 4.0, 5.0};
    for (auto x : v) {
        cout << x << ", "; // problem: extra comma
    }
    cout << endl;

    for(auto it = v.begin(); it != v.end(); it++) { //iterators
        cout << *it;
        if ((it+1) != v.end()) {
            cout << ", ";
        }
    }
    cout << endl;

    // C++ vector holds contiguous memory. Then, we can
    // use operator[] and perform C-style for-loop
    auto sz = v.size();
    for (auto i = 0; i < sz; i++) {
        cout << v[i];
        if ((i+1) < sz) {
            cout << ", ";
        }
    }
    cout << endl;

    // FMT offers a more concise format (C++23)
    print("{}", join(v, ", "));
    print("\n");
}

```

# C++ IO

How to print elements  
of a container (vector,  
list and tuples)?

Big difference if (1)  
memory is (non)  
contiguous or (2)  
container is iterable

**std::vector<T>** easier

```

whovian@phy504:~/host/CPP$ g++ io6.c -o io6 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io6
1, 2, 3, 4, 5,
1, 2, 3, 4, 5
1, 2, 3, 4, 5
1, 2, 3, 4, 5

```

```
#include <iostream>
#include <fmt/core.h>
#include <list>
#include <fmt/ranges.h> // fmt::join
using fmt::print; using fmt::join; using std::endl;
using std::list; using std::cout;

int main ()
{
    // remember that list is double linked list
    list<double> v = {1.0, 2.0, 3.0, 4.0, 5.0};
    for (auto x : v) {
        cout << x << ", ";
    }
    cout << endl;

    for(auto it = v.begin(); it != v.end(); it++) { //iterators
        cout << *it;
        //if ((it+1) != v.end()) { // you can't make arithmetic
            // list iterators (noncontiguous
            // memory
        if (++it != v.end()) {
            cout << ", ";
        }
        it--; // these operators are ok (C++ doesn't behave like C!)
    }
    cout << endl;

    // C++ list holds noncontiguous memory. Then, we cannot (!!)
    // use operator[] and perform C-style for-loop

    // FMT offers a more concise format (C++23)
    print("{}", join(v, ", "));
    print("\n");
}
```

# C++ IO

How to print elements of a container?

**std::list<T>** is non-contiguous. No arithmetic allowed on iterators but **i++**, **i--** ok because they mean next/previous

```
whovian@phy504:~/host/CPP$ g++ io7.c -o io7 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io7
1, 2, 3, 4, 5,
1, 2, 3, 4, 5
1, 2, 3, 4, 5
```

```
#include <iostream>
#include <fmt/core.h>
#include <fmt/ranges.h>
#include <tuple>
using fmt::print; using std::string;
using fmt::join; using namespace std;

int main ()
{
    tuple<double, string, int> v1 = {2.0, "Vivian", 1000};

    // tuple is noniterable - complicated to print all
    // elements using C++ streams (requires an advanced
    // feature named variadic generic lambdas/templates)

    // We won't cover variadics in this lecture

    // FMT offers a more concise/easier format (C++23)
    print("{}", fmt::join(v1, ", "));
    print("\n");

    // C++17 - use initialization to guess template parameters
    tuple v2 = {10.0, string("VivianMiranda"), 2000};

    print("{}", fmt::join(v2, ", "));
    print("\n");
}
```

```
whovian@phy504:~/host/CPP$ g++ io8.c -o io8 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io8
2, Vivian, 1000
10, VivianMiranda, 2000
```

# C++ IO

How to print  
elements of a  
container (vector,  
list and tuples)?

Big difference if  
(1) memory is  
(non) contiguous  
or (2) container is  
iterable

```

#include <fstream>
#include <iostream>
#include <list>
#include <fmt/core.h>
#include <iomanip>    // setw, setfill, setprecision
using namespace std;

int main () {
    list<double> v = {1.0, 2.0, 3.0, 4.0, 5.0};

    fstream file;
    file.open("test_io9.txt", ios::out);
    if(!file.is_open()) {
        cout << "Error in creating file!!!" << endl;
        return 1;
    }

    for (auto x : v) { // only change is cout <-> file
        file << right << setfill('*') <<
            setw(21) << setprecision(10) <<
            fixed << x << endl;
    }
    file.close();

    // via FMT library (part I using format)
    list<double> v2 = {20.0, 40.0, 60.0, 80.0, 100.0};

    file.open("test_io9.txt", ios::app); // append
    if(!file.is_open()) {
        cout << "Error oppening file" << endl;
        return 1;
    }

    for (auto x : v2) {
        file << fmt::format("{:>21.10f}", x) << endl;
    }
    file.close();
}

```

# C++ IO

## How to write to files?

```

whovian@phy504:~/host/CPP$ g++ io9.c -o io9 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io9
whovian@phy504:~/host/CPP$ more test_io9.txt
*****1.0000000000
*****2.0000000000
*****3.0000000000
*****4.0000000000
*****5.0000000000
*****20.0000000000
*****40.0000000000
*****60.0000000000
*****80.0000000000
*****100.0000000000

```

**streams:** users only need to  
replace **std::cout** with  
**std::fstream**

Remember that streams are  
compatible with  
**fmt::format**

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name;
    int age;
    cin >> name >> age;

    cout << "Name : " << name << endl;
    cout << "Age : " << age << endl;

    for (int x; !(cin>>x).fail();)
    {
        cout << "x : " << x << endl;
    }

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ io10.c -o io10 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io10
VivianMiranda 37
Name : VivianMiranda
Age : 37
1
x : 1
2
x : 2
3
x : 3
xw
```

## C++ IO

How to read from the terminal (and files)?

# C++ IO

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name;
    int age;
    cin >> name >> age;

    cout << "Name : " << name << endl;
    cout << "Age : " << age << endl;

    for (int x; !(cin>>x).fail();)
    {
        cout << "x : " << x << endl;
    }

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ io10.c -o io10 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io10
VivianMiranda 37
Name : VivianMiranda
Age : 37
1
x : 1
2
x : 2
3
x : 3
xw
```

How to read from the terminal (and files)?

Invert from **<<** & **cout** (output) to **>>** & **cin** (input)

```
#include <fstream>
#include <iostream>
#include <string>
#include <iomanip> //setw, setfill, setprecision
using namespace std;

int main ()
{
    fstream file;
    file.open("test_io11.txt", ios::in);
    if(!file.is_open()) {
        cout << "Error reading the file!!!" << endl;
        return 1;
    }

    string x;
    int y;
    file >> x >> y;

    cout << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << x << endl;

    cout << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << y << endl;

    return 0;
}
```

# C++ IO

How to read from the terminal (and files)?

Invert from <<  
(output) to >> (input)

Don't forget the  
**ios::in** flag

```
whovian@phy504:~/host/CPP$ more test_io11.txt
VivianMiranda 37
whovian@phy504:~/host/CPP$ g++ io11.c -o io11 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io11
*****VivianMiranda
*****37
```

# C++ IO

How to read from  
the terminal (and  
files)?

Instead of using  
**ios::in/out** flags,  
you can also use  
**std::ifstream** or  
**std::ofstream**

```
#include <fstream>
#include <iostream>
#include <string>
#include <iomanip> //setw, setfill, setprecision
using namespace std;

int main ()
{
    // Instead of using fstream, C++ offers ifstream
    // and ofstream as well. What is the difference?
    // You don't need the .open( , ios::flag) line
    ifstream file("test_io11.txt"); // open via constructor
    if(!file.is_open()) {
        cout << "Error reading the file!!!" << endl;
        return 1;
    }

    string x;
    int y;
    file >> x >> y;

    cout << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << x << endl;

    cout << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << y << endl;

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ io13.c -o io13 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io13
*****VivianMiranda
*****37
```

```

#include <sstream> // stringstream
#include <iostream>
#include <string>
#include <iomanip> // setw, setfill, setprecision
#include <fmt/core.h>
using namespace std;

int main ()
{
    std::stringstream ss;

    ss << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << 12.3 << " and ";
    ss << left << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << 15.6;

    string s1 = ss.str(); // get the string
    cout << s1 << endl;

    // via FMT library (and C++23 standard)
    string s2 = fmt::format(
        "{:.*>21.10f} and {:.*<21.10f}", 12.3, 15.6);
    cout << s2 << endl;

    return 0;
}

```

# C++ IO

```

whovian@phy504:~/host/CPP$ g++ io12.c -o io12 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io12
*****12.3000000000 and 15.6000000000*****
*****12.3000000000 and 15.6000000000*****

```

C++ offers streams  
for strings (C++20  
offers **std::format**)

Works like any  
other stream, but it  
does not output to  
terminal/file  
(can work as buffer)

# C++ IO

```
#include <sstream> // stringstream
#include <fstream>
#include <iostream>
#include <string>
#include <iomanip> //setw, setfill, setprecision

using namespace std;

int main ()
{
    ifstream file("test_io11.txt");
    if(!file.is_open()) {
        cout << "Error reading the file!!!" << endl;
        return 1;
    }

    // read the entire file into memory
    std::stringstream buffer;
    buffer << file.rdbuf(); // read buffer
    file.close();

    std::string file_content = buffer.str();
    if (file_content.empty()) {
        cout << "file empty" << endl;
    }
    cout << file_content << endl;

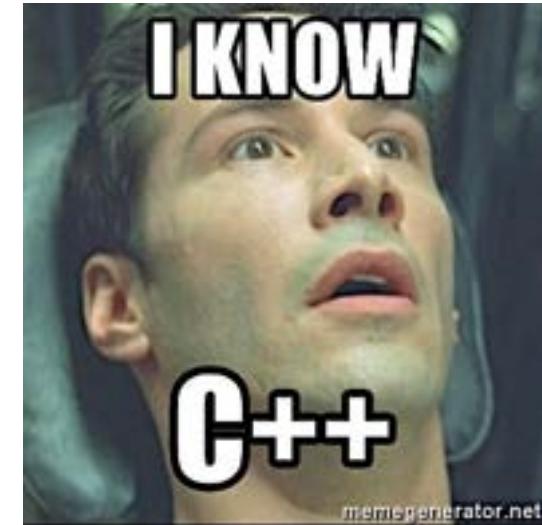
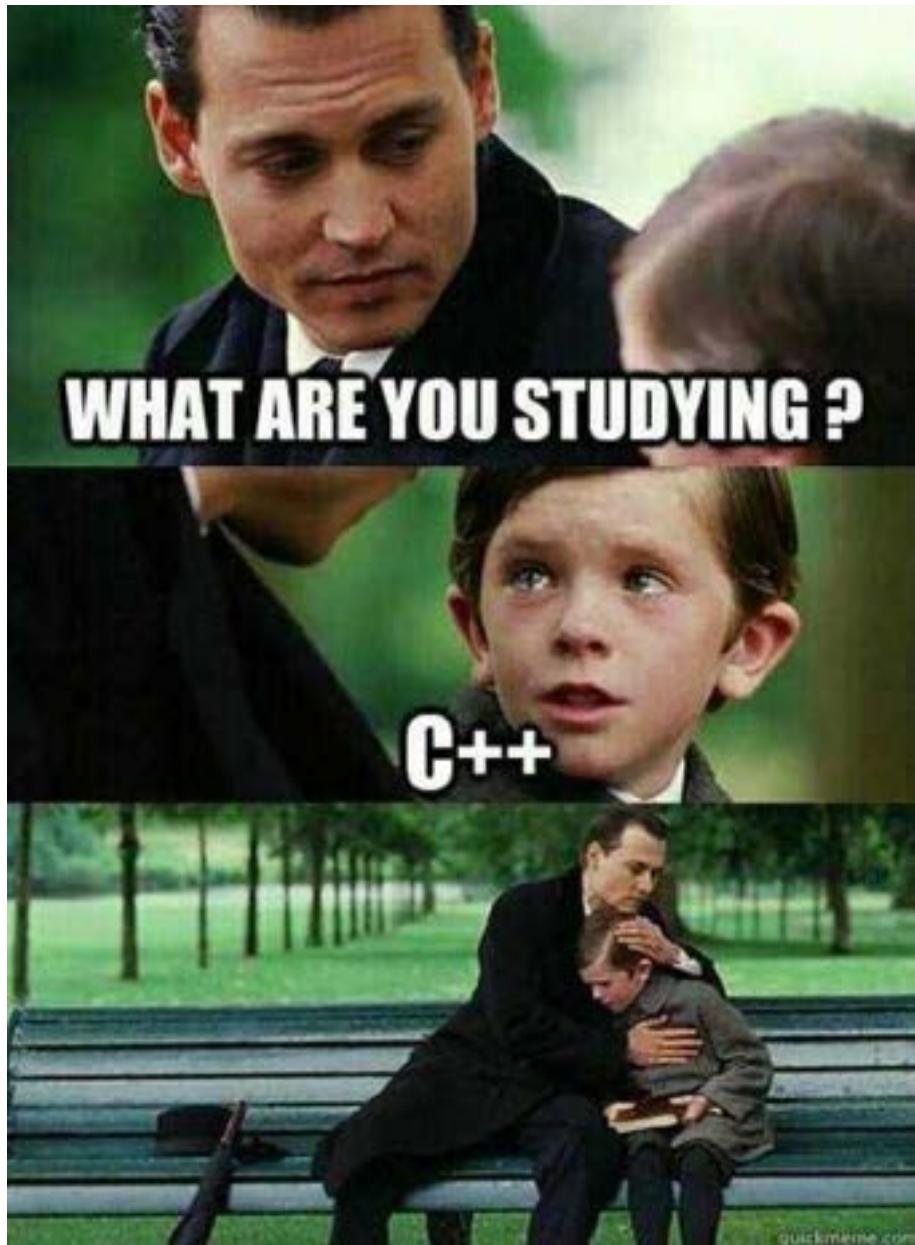
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ io14.c -o io14 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io14
VivianMiranda 37
```

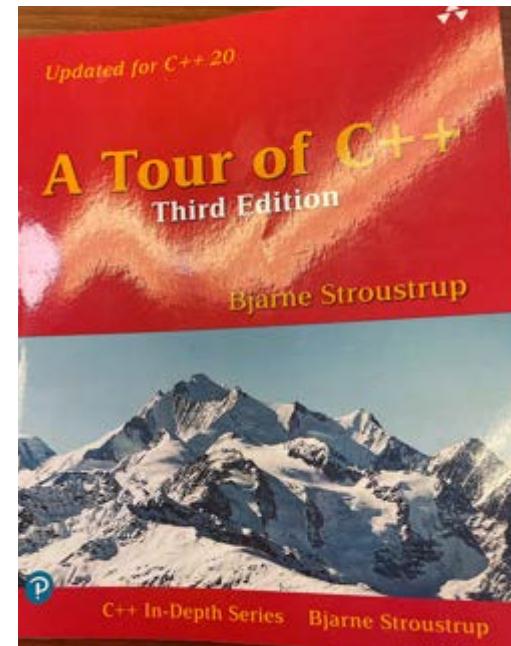
C++ offers streams for strings (C++20 offers **std::format**)

We can use C++ **std::stringstream** to read entire file on RAM buffer

# Lecture 30: C++ part VI



## Suggested Literature



```
#include <string>
#include <iostream>

using namespace std;

enum class Human : int
{
    Vivian,
    Beyonce,
    Rihanna
};

enum class Professor : int
{
    Vivian, // Not a problem to have same name
    Wayne,
    Ioav
};

int main ()
{
    Human x = Human::Rihanna;

    if (x == Human::Rihanna)
        cout << "Human is Rihanna" << endl;
    else
        cout << "Human is not Rihanna" << endl;

    Professor y = Professor::Vivian;

    if (y == Professor::Vivian)
        cout << "Professor is Vivian" << endl;
    else
        cout << "Professor is not Vivian" << endl;

    return 0;
}
```

# C++ Enum Classes

```
whovian@phy504:~/host/CPP$ g++ enumclass1.cpp -o enumclass1 -std=c++20
whovian@phy504:~/host/CPP$ ./enumclass1
Human is Rihanna
Professor is Vivian
```

**enum class** solves the main problem of enumerators in C.

Variables inside C enumerators are global. Big problem: you can't have two enumerators with variables that share the same name

# C++ Classes Part I

```
class Base {  
public:  
    void print() {  
        // code  
    }  
};  
  
class Derived : public Base {  
public:  
    void print() {  
        // code  
    }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```



```
#include <array>
#include <string>
#include <list>
#include <iostream>
using namespace std;

// constexpr = compile time constant (C++11 feature)
constexpr int NDigitsSSN = 9;
constexpr int NBirthday = 3; // birthday month, day, year
constexpr int NDigitsYear = 4; // Force Year to be 4 digits
constexpr int NCountries = 195; // number of countries
constexpr int CurrentYear = 2023;

enum class Genders : int
{ // C++ version of enumerators (safer than C version)
    Female,
    Male,
    NB
};

class Human // Simpler C-Style class
{
public:
    int age_;
    int ssn_ = -1;           // default value (flag not set)
    Genders gender_;
    double height_ = -1.0;   // default value (flag not set)
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_; // We may have multiple nationalities
    list<string> name_;     // We have name, surname
private:
    // none
};

int main() {
    Human x;
    x.age_ = 37;
    x.name_ = {"Vivian", "Miranda"};
    cout << x.age_ << endl;
    return 0;
}
```

# C++ Classes

## C++ Class v0.1

**public:** / **private:**  
access control

C-Style classes are only  
possible if all variables  
is set to have **public**  
access

In C++, default for **class**  
is **private** access

```
class Human
{
public:
    // Constructor - we require name when
    // creating an instance of the class
    // No Human without defined name
    Human(list<string> name) :
        name_(name)
    {
    }
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    // none
};

int main() {
    // Human x; // This line does not compile
    Human x({"Vivian", "Miranda"});
    x.age_ = 37;
    cout << x.age_ << endl;
    return 0;
}
```

Constructor has the  
same name of the class

# C++ Classes

## C++ Class v0.2

### Constructors

tells the compiler how  
an instance of the class  
should be constructed

Example: now Human  
instance should be  
created without defined  
name

```
class Human {
public:
    Human(list<string> name) :
        name_(name)
    {
    }
    Human(const string name)
    {
        boost::split(this->name_, name,
                    boost::is_any_of(" \t"),
                    boost::token_compress_on);
    }
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    // none
};

int main() {
    // Human x; // This line does not compile
    // Human x({"Vivian", "Miranda"}); // This line doesn't
                                       // compile (ambiguous)
    Human x(list<string>{"Vivian", "Miranda"});
    x.age_ = 37;
    cout << x.age_ << endl;

    Human y(string{"Vivian Miranda"});
    y.age_ = 37;
    cout << y.age_ << endl;

    return 0;
}
```

# C++ Classes

## C++ Class v0.3

We can have multiple  
**constructors** in C++

We can use a new  
**constructor** to convert  
from **std::string** to  
**std::list<string>** using a  
code snippet from  
previous lecture

**this->** is optional (you  
should always include it)

# C++ Classes

## C++ Class v0.3

We can have multiple **constructors** in C++, but more is not always better (choose the easier API)

```
int main() {
    // Human x; // This line does not compile
    // Human x({"Vivian", "Miranda"}); // This line doesn't
                                         // compile (ambiguous)
    Human x(list<string>{"Vivian", "Miranda"});
    x.age_ = 37;
    cout << x.age_ << endl;

    Human y(string{"Vivian Miranda"});
    y.age_ = 37;
    cout << y.age_ << endl;

    return 0;
}
```

```
:whovian@phy504:~/host/CPP$ g++ class3.cpp -o class3 -std=c++20
class3.cpp: In function 'int main()':
class3.cpp:48:32: error: call of overloaded 'Human(<brace-enclosed initializer list>)' is ambiguous
  48 |     Human x({"Vivian", "Miranda"});
                 ^
class3.cpp:29:3: note: candidate: 'Human::Human(std::string)'
  29 |     Human(const string name)
                 ^
class3.cpp:25:3: note: candidate: 'Human::Human(std::__cxx11::list<std::__cxx11::basic_string<char> >)'
  25 |     Human(list<string> name)
                 ^
```

# C++ Classes

## C++ Class v0.4

Choose the  
easier API  
when designing  
**constructors**

```
class Human {
public:
    Human(const string name)
    {
        boost::split(this->name_, name,
                    boost::is_any_of(" \t"),
                    boost::token_compress_on);
    }
    Human(const string name, array<int, NBirthday> birthday) :
        Human(name) // Construtor delegation avoids code repetition
    {
        this->birthday_ = birthday;
        this->age_ = CurrentYear - this->birthday_.at(2);
    }
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    // none
};

int main() {
    Human x("Vivian Miranda", {02,28,1986});
    cout << x.age_ << endl;
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ class4.cpp -o class4 -std=c++20
```

```
whovian@phy504:~/host/CPP$ ./class4
```

```
class Human {
public:
    Human(const string name,
          array<int, NBirthday> birthday = {0,0,0})
    {
        boost::split(this->name_, name,
                    boost::is_any_of(" \t"),
                    boost::token_compress_on);

        this->birthday_ = birthday;
        this->age_ = CurrentYear - this->birthday_.at(2);
    }
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    // none
};

int main() {
    Human x("Vivian Miranda", {02,28,1986});
    cout << x.age_ << endl;

    Human y("Vivian Miranda");
    y.age_ = 37;
    cout << y.age_ << endl;
    return 0;
}
```

# C++ Classes

## C++ Class v0.5

In this example, coding multiple **constructors** were not really needed

Use default arguments instead

```
whovian@phy504:~/host/CPP$ nano class5.cpp
whovian@phy504:~/host/CPP$ g++ class5.cpp -o class5 -std=c++20
whovian@phy504:~/host/CPP$ ./class5
37
37
```

```
class Human {
public:
    Human(const string name,
          array<int, NBirthday> birthday = {0,0,-1}) :
        birthday_(birthday),
        is_birthday_set_(false),
        age_(CurrentYear - birthday.at(2))
    {
        boost::split(this->name_, name,
                    boost::is_any_of(" \t"),
                    boost::token_compress_on);

        if(!(*this->age_ > CurrentYear)) {
            this->is_birthday_set_ = true;
        }
    }
    void PrintBirthday () { // member function
        auto& x = *this->birthday_; // just variable alias (& = ref)

        if(!this->is_birthday_set_) {
            spdlog::critical("Birthday not set");
            exit(1);
        }
        fmt::print("Birthday: {:02d},{:02d},{:04d}\n",x[0],x[1],x[2]);
    }
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    bool is_birthday_set_; // can only be accessed by member function
};

int main() {
    Human x("Vivian Miranda", {02,28,1986});
    x.PrintBirthday();
    // Problem: if you change birthday_ manually, you can't
    // use PrintBirthday() because we can't access is_birthday_set_
    Human y("Vivian Miranda");
    y.birthday_ = {02,28,1986};
    y.PrintBirthday();
}
```

# C++ Classes

## C++ Class v0.6

C++ class allow the definition of **member functions**

**public:** / **private:** access makes more sense. Private variables can only be accessed via **member functions**

```
whovian@phy504:~/host/CPPS g++ class6.cpp -o class6 -std=c++20 -lfmt
whovian@phy504:~/host/CPPS ./class6
Birthday: 02,28,1986
[2023-04-21 15:42:43.684] [critical] Birthday not set
```

```
class Human {
public:
    Human(const string name,
          array<int, NBirthday> birthday = {0,0,-1});

    void PrintBirthday();
private: // Lets move all variables to be private
    bool is_birthday_set_;
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
};

Human::Human(const string name, array<int, NBirthday> birthday) : :
    birthday_(birthday),
    is_birthday_set_(false),
    age_(CurrentYear - birthday.at(2))
{
    boost::split(this->name_, name,
                boost::is_any_of(" \t"),
                boost::token_compress_on);

    if(!(this->age_ > CurrentYear)) {
        this->is_birthday_set_ = true;
    }
}

void Human::PrintBirthday () {
    auto& x = this->birthday_;
    if(!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    fmt::print("Birthday: {:02d},{:02d},{:04d}\n",x[0],x[1],x[2]);
}
```

# C++ Classes

## C++ Class v0.7

Better to strictly control access to  
**member variables** (via  
**member functions**)

Also, better for readability to just declare **member functions** inside the C++ **class**

```
class Human {
public:
    Human(const string name, array<int, 3> bd = {0,0,-1});

    void SetBirthday(const int month, const int day,
                     const int year);

    void SetBirthday(array<int, 3> x);

    void SetGender(const Genders x);

    void AddNationality(const int x);

    array<int, 3> GetBirthday() const;

    Genders GetGender() const;

    int GetAge() const;

    list<int> GetNationality() const;

    void Print() const;
private:
    bool is_birthday_set_ = false;
    bool is_gender_set_ = false;
    bool is_min_one_nationality_set_ = false;
    int age_ = 0;
    Genders gender_;
    array<int, 3> birthday_ = {0, 0, -1};
    list<int> nationality_;
    list<string> name_;
};
```

# C++ Classes

## C++ Class v0.8

Getting closer to the v1.0 release of the Human class!

Functions that do not change member variables should have the keyword **const** (const correctness)

```
Human::Human(const string name, array<int, 3> birthday) :  
    birthday_(birthday),  
    age_(CurrentYear - birthday.at(2)) {  
    boost::split(this->name_, name, boost::is_any_of(" \t"),  
                boost::token_compress_on);  
    if(!(this->age_ > CurrentYear)) {  
        this->is_birthday_set_ = true;  
    }  
}  
  
void Human::SetBirthday(const int month, const int day,  
                       const int year) {  
    this->SetBirthday(array<int, 3>{month, day, year});  
}  
  
void Human::SetBirthday(array<int, 3> x) {  
    if (x[0] < 0 || x[0] > 12)  
    {  
        spdlog::critical("Birthday Month Invalid");  
        exit(1);  
    }  
    if (x[2] > CurrentYear)  
    {  
        spdlog::critical("Birthday Year Invalid");  
        exit(1);  
    }  
    if (x[1] < 0 || x[1] > 31)  
    {  
        spdlog::critical("Birthday Day Invalid");  
        exit(1);  
    }  
    this->birthday_ = x;  
    this->is_birthday_set_ = true;  
    this->age_ = CurrentYear - x[2];  
}  
  
void Human::SetGender(const Genders x) {  
    this->gender_ = x;  
    this->is_gender_set_ = true;  
}  
  
void Human::AddNationality(const int x) {  
    this->nationality_.push_back(x);  
    this->is_min_one_nationality_set_ = true;  
}
```

# C++ Classes

## C++ Class v0.8

Note that for member function bodies outside the class, the

**Human::** keyword must precede the function name

Member functions don't exist in isolation. They are intrinsically tied to the class

```
array<int, 3> Human::GetBirthday() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->birthday_;
}

Genders Human::GetGender() const {
    if (!this->is_gender_set_) {
        spdlog::critical("Gender not set");
        exit(1);
    }
    return this->gender_;
}

int Human::GetAge() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->age_;
}

list<int> Human::GetNationality() const {
    if (!this->is_min_one_nationality_set_) {
        spdlog::critical("No Nationality set");
        exit(1);
    }
    return this->nationality_;
}

void Human::Print() const {
    fmt::print("Name: ");
    for (auto x : this->name_) {
        fmt::print("{0} ", x);
    }
    fmt::print("\n");
    auto& x = this->birthday_; // just variable alias (& = ref)
    if(!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    fmt::print("Birthday: {:02d},{:02d},{:04d}\n", x[0],x[1],x[2]);
}
```

# C++ Classes

## C++ Class v0.8

One advantage of **get/set** type of member functions is they provide opportunities for the developer to include extra checks

Function overloading via the number (and/or type) of arguments works for member functions as well

# C++ Classes

## C++ Class v0.8

### Const correctness.

```
array<int, 3> Human::GetBirthday() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->birthday_;
}

Genders Human::GetGender() const {
    if (!this->is_gender_set_) {
        spdlog::critical("Gender not set");
        exit(1);
    }
    return this->gender_;
}

int Human::GetAge() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->age_;
}

list<int> Human::GetNationality() const {
    if (!this->is_min_one_nationality_set_) {
        spdlog::critical("No Nationality set");
        exit(1);
    }
    return this->nationality_;
}

void Human::Print() const {
    fmt::print("Name: ");
    for (auto x : this->name_) {
        fmt::print("{0} ", x);
    }
    fmt::print("\n");
    auto& x = this->birthday_; // just variable alias (& = ref)
    if(!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    fmt::print("Birthday: {:02d},{:02d},{:04d}\n", x[0],x[1],x[2]);
}
```

const copies and const references to Human class can only access functions that haven't the **const** keyword in their declaration (except constructor, of course)

```
array<int, 3> Human::GetBirthday() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->birthday_;
}

Genders Human::GetGender() const {
    if (!this->is_gender_set_) {
        spdlog::critical("Gender not set");
        exit(1);
    }
    return this->gender_;
}

int Human::GetAge() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->age_;
}

list<int> Human::GetNationality() const {
    if (!this->is_min_one_nationality_set_) {
        spdlog::critical("No Nationality set");
        exit(1);
    }
    return this->nationality_;
}

void Human::Print() const {
    fmt::print("Name: ");
    for (auto x : this->name_) {
        fmt::print("{0} ", x);
    }
    fmt::print("\n");
    auto& x = this->birthday_; // just variable alias (& = ref)
    if(!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    fmt::print("Birthday: {:02d},{:02d},{:04d}\n", x[0],x[1],x[2]);
}
```

# C++ Classes

## C++ Class v0.8

### Const Correctness.

const copies and const references to Human class can only access functions that have **const** in their declaration (except constructor, of course)

Const member functions can still affect mutable variables

# C++ Classes

```
int main() {
    const Human x("Vivian Miranda", {02,28,1986});

    // Here you can't use set functions!!!

    cout << x.GetAge() << endl; // ok

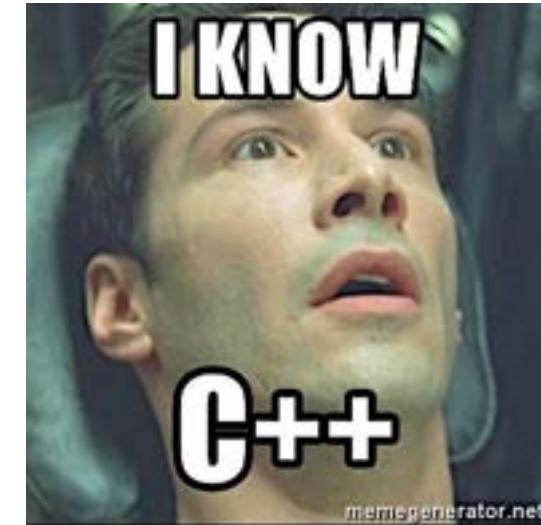
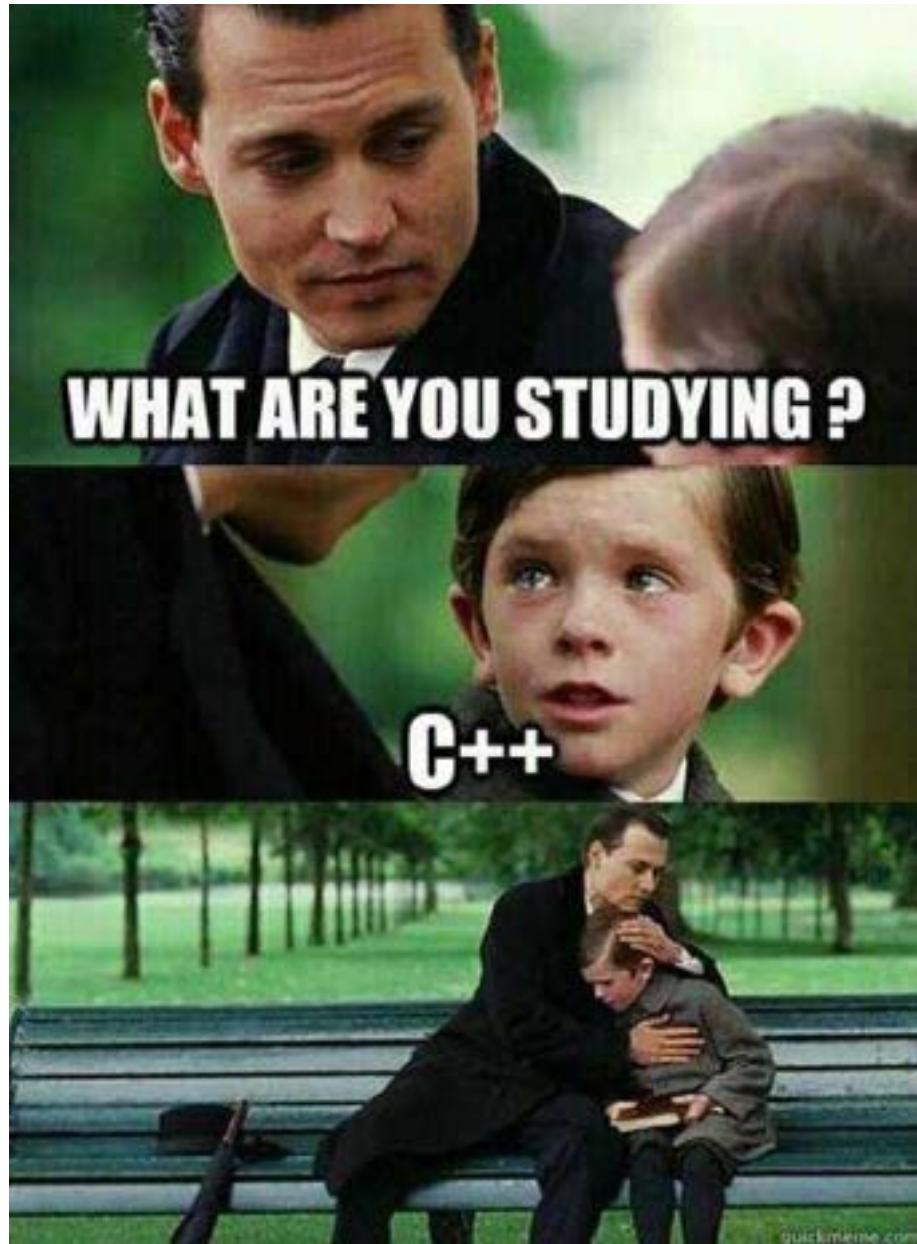
    x.Print(); // ok

    //x.AddNationality(55); // compilation error
}
```

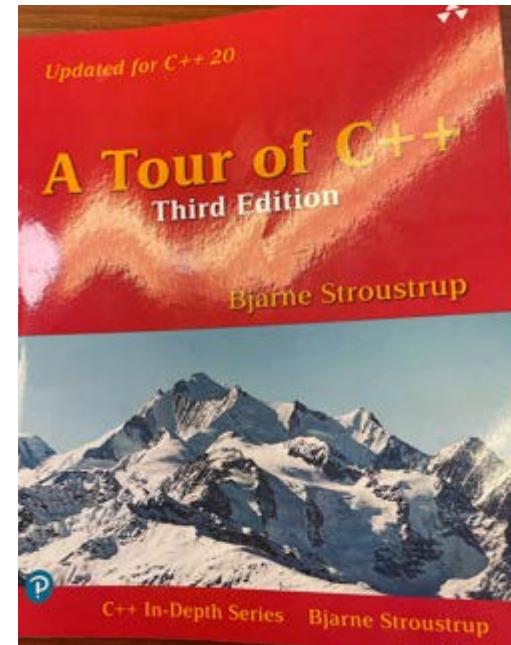
```
whovian@phy504:~/host/CPP$ g++ class8.cpp -o class8 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./class8
37
Name: Vivian Miranda
Birthday: 02,28,1986
```

```
whovian@phy504:~/host/CPP$ g++ class8.cpp -o class8 -std=c++20 -lfmt
class8.cpp: In function 'int main()':
class8.cpp:154:19: error: passing 'const Human' as 'this' argument discards qualifiers [-fpermissive]
  154 |     x.AddNationality(55); // compilation error
      |     ~~~~~^~~~~~
class8.cpp:94:6: note:   in call to 'void Human::AddNationality(int)'
  94 | void Human::AddNationality(const int x) {
      |     ^~~~~~
```

# Lecture 31: C++ part VII



## Suggested Literature



# C++ Classes Part II

```
class Base {  
public:  
    void print() {  
        // code  
    }  
};  
  
class Derived : public Base {  
public:  
    void print() {  
        // code  
    }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```



```
#include <string>
#include <iostream>
using namespace std;

class Base {
public:
    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintBase: " << tmp << endl;
    }
private:
    int x_ = 2;
};

class Derived : public Base {
public:
    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintDerived: " << tmp << endl;
    }
private:
    int x_ = 10;
};

int main ()
{
    Base y;
    y.print(100);

    Derived x;
    x.print(100);
}
```

# Class Inheritance

Classes can have parents, grandparents and so on...

**Public** inheritance implies that the child class cannot access private member functions. E.g., we can define independent **int x\_**

One child class can have multiple parents

```
whovian@phy504:~/host/CPP$ g++ class9.cpp -o class9 -std=c++20
whovian@phy504:~/host/CPP$ ./class9
PrintBase: 98
PrintDerived: 90
```

```
#include <string>
#include <iostream>
using namespace std;

class Base {
    int x_ = 2; // default visibility is private
public:
    void print(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
};

class Derived : public Base {
    int x_ = 10; // default visibility is private
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main () {
    Derived x;
    x.print(100);

    // const here is mandatory (compiler error)
    const Base& y = static_cast<Base>(x);
    y.print(100);

    Base* z = static_cast<Base*>(&x);
    z->print(110); // pointers use -> instead of .

    Base w = static_cast<Base>(x); // copy (only parent)!
    w.print(120);
}
```

# Class Inheritance

Child and parent classes can have member functions with the same name and arg. types (no overloading).

```
whovian@phy504:~/host/CPP$ g++ class10.cpp -o class10 -std=c++20
whovian@phy504:~/host/CPP$ ./class10
PrintDerived: 90
PrintBase: 98
PrintBase: 108
PrintBase: 118
```

Access to the parent's version happens via casting

```
#include <string>
#include <iostream>
using namespace std;

class Base {
    int x_ = 2; // default visibility is private
public:
    void print(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
};

class Derived : public Base {
    int x_ = 10; // default visibility is private
public:
    void print2(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main ()
{
    Derived x;
    x.print(100);
    x.print2(100);
}
```

# Class Inheritance

If member functions have different name, then no explicit casting is needed

```
whovian@phy504:~/host/CPP$ g++ class11.cpp -o class11 -std=c++20
whovian@phy504:~/host/CPP$ ./class11
PrintBase: 98
PrintDerived: 90
```

```
#include <string>
#include <iostream>
using namespace std;

class Mom {
    int x_ = 2; // default visibility is private
public:
    void print(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
};

class Dad {
    int x_ = 10; // default visibility is private
public:
    void print2(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
};

class Derived : public Mom, public Dad {
    int x_ = 50; // default visibility is private
public:
    void print3(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main ()
{
    Derived x;
    x.print(100);
    x.print2(100);
    x.print3(100);
}
```

# Class Inheritance

One child class can have multiple parents

Again, if member functions have different names, then no explicit casting is needed

All **private** variables are independent

```
whovian@phy504:~/host/CPP$ g++ class12.cpp -o class12 -std=c++20
whovian@phy504:~/host/CPP$ ./class12
PrintBase: 98
PrintBase: 90
PrintDerived: 50
```

```
#include <string>
#include <iostream>
using namespace std;

class Base {
public:
    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintBase: " << tmp << endl;
    }
private:
    int x_ = 2;
};

class Derived : public Base {
public:
    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintDerived: " << tmp << endl;
    }
private:
    int x_ = 10;
};

int main ()
{
    Base y;
    y.print(100);

    Derived x;
    x.print(100);
}
```

# Class Inheritance

## Constructors (non-default)

What happens if the user defines a **non-default constructor** (constructor w/ no args/parameters)?

Remember that if the user defines a constructor that requires an argument, the **default constructor** is not accessible (unless explicitly coded)

```

#include <string>
#include <iostream>
using namespace std;

class Base {
    int x_ = 2;
public:
    Base(const double x) :
        x_(x)
    {}
    void print(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
};

class Derived : public Base {
    int x_ = 10;
public:
    Derived(const double x, const double y) :
        Base(x), ← Call Base from Derived
        x_(y)
    {}
    void print2(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main ()
{
    Derived x(10, 30);
    x.print(100);
    x.print2(100);
}

```

# Class Inheritance

Remember: you can't access **Base** private members from **Derived**

Therefore, we need to delegate constructors (call the **Base** constructor from the **Derived** constructor)

```

whovian@phy504:~/host/CPP$ nano class13.cpp
whovian@phy504:~/host/CPP$ g++ class13.cpp -o class13 -std=c++20
whovian@phy504:~/host/CPP$ ./class13
PrintBase: 90
PrintDerived: 70

```

```

#include <string>
#include <iostream>
using namespace std;

class Mom {
    int x_ = 2;
public:
    void print(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
    Mom(const double x) :
        x_(x) {}
};

class Dad {
    int x_ = 10;
public:
    void print2(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
    Dad(const double x) :
        x_(x) {}
};

class Derived : public Mom, public Dad {
    int x_ = 50;
public:
    Derived(const double x, const double y, const double z) :
        Mom(x),
        Dad(y),
        x_(z) {}

    void print3(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main () {
    Derived x(10, 20, 30);
    x.print(100);
    x.print2(100);
    x.print3(100);
}

```

# Class Inheritance

Remember: you can't access **Base** private members from **Derived**

Therefore, we need to delegate constructors (call the **Base** constructor from the **Derived** constructor)

```

whovian@phy504:~/host/CPP$ g++ class14.cpp -o class14 -std=c++20
whovian@phy504:~/host/CPP$ ./class14
PrintBase: 90
PrintBase: 80
PrintDerived: 70

```

```
#include <iostream>
using namespace std;

class Base {
    int x_ = 2;
public:
    virtual void print(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
    void print2(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
};

class Derived : public Base {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
    void print2(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main () {
    Derived x;
    // first time we use dynamic cast
    Base* y = dynamic_cast<Base*>(&x);

    // 1) Virtual function (runtime binding)
    y->print(100);
    // 2) Non-virtual function (compilation binding)
    y->print2(100);
}
```

# Class Inheritance

What if you want to cast **Derived** to **base** (at the pointer level) but continue to use the **Derived** version of the member function?

Then you need to make the member functions **virtual**

```
whovian@phy504:~/host/CPP$ g++ class15.cpp -o class15 -std=c++20
whovian@phy504:~/host/CPP$ ./class15
PrintDerived: 90
PrintBase: 98
```

```

#include <iostream>
using namespace std;

class Base {
    int x_ = 2;
public:
    virtual void print(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
};

class Derived : public Base {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

class Derived2 : public Base {
    int x_ = 50;
public:
    void print(const double x) const {
        cout << "PrintDerived2: " << x->x_ << endl;
    }
};

int main () {
    Derived x;

    Base* y = dynamic_cast<Base*>(&x); // good cast
    if (y == nullptr)
        cout << "dynamic_cast<Base*> failed" << endl;

    Derived2* z = dynamic_cast<Derived2*>(&x); // bad cast
    if (z == nullptr)
        cout << "dynamic_cast<Derived2*> failed" << endl;
}

```

```

whovian@phy504:~/host/CPP$ g++ class16.cpp -o class16 -std=c++20
class16.cpp: In function 'int main()':
class16.cpp:33:17: warning: 'dynamic_cast<class Derived2*>(Derived x)' can never succeed
  33 |     Derived2* z = dynamic_cast<Derived2*>(&x); // bad cast
      |
      ^~~~~
whovian@phy504:~/host/CPP$ ./class16
dynamic_cast<Derived2*> failed

```

# Class Inheritance

What happens when  
**dynamic\_cast** fails  
(illegal casting attempt)?

returns **nullptr**

Compiler may catch as  
warning (see above)

```
#include <iostream>
using namespace std;

class Mom {
    int x_ = 2;
public:
    virtual void print(const double x) const {
        cout << "PrintMom: " << x->x_ << endl;
    }
};

class Child : public Mom {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintChild: " << x->x_ << endl;
    }
};

class GrandChild : public Child {
    int x_ = 20;
public:
    void print(const double x) const {
        cout << "PrintGrandChild: " << x->x_ << endl;
    }
};

int main () {
    Child x;
    Mom* y = dynamic_cast<Mom*>(&x);
    y->print(100);

    GrandChild x2;
    Mom* y2 = dynamic_cast<Mom*>(&x2);
    y2->print(100);
}
```

# Class Inheritance

What happens with  
**dynamic\_cast** when  
you have child and  
grandchild?

Only the **Mom** class  
needs the word **virtual**

```
whovian@phy504:~/host/CPP$ g++ class17.cpp -o class17 -std=c++20
whovian@phy504:~/host/CPP$ ./class17
PrintChild: 90
PrintGrandChild: 80
```

```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main () {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);
    y->print(100);
    // Base x2; Illegal line of C++
}
```

# Class Inheritance

## Abstract **Base** Class

Abstract base class  
cannot be created,  
only dynamically  
casted from  
derived class

```
whovian@phy504:~/host/CPP$ g++ class18.cpp -o class18 -std=c++20
whovian@phy504:~/host/CPP$ ./class18
PrintDerived: 90
```

```
#include <string>
#include <iostream>

using namespace std;

class Base {
protected:
    int x_ = 2;
public:
    void print(const double x) const {
        cout << "PrintBase: " << x->x_ << endl;
    }
};

class Derived : public Base {
public:
    void print2(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main ()
{
    Derived x;
    x.print(100);
    x.print2(100);
}
```

Class  
Inheritance  
Between **public:**  
and **private:** - is  
there a middle  
ground?

Yes! **protected:**  
(not public, but  
accessible by  
**Derived** class)

```
whovian@phy504:~/host/CPP$ g++ class19.cpp -o class19 -std=c++20
whovian@phy504:~/host/CPP$ ./class19
PrintBase: 98
PrintDerived: 98
```

```

#include <string>
#include <iostream>
using namespace std;

class God {
protected:
    int x_ = 2;
public:
    void print(const double x) const {
        cout << "PrintGod: " << x->x_ << endl;
    }
};

class Mom : public God {
public:
    void print(const double x) const {
        cout << "PrintMom: " << x->x_ << endl;
    }
};

class Dad : public God {
public:
    void print(const double x) const {
        cout << "PrintDad: " << x->x_ << endl;
    }
};

class Kid : public Mom, public Dad {
public:
    void print(const double x) const {
        cout << "PrintKid: " << x->x_ << endl;
    }
};

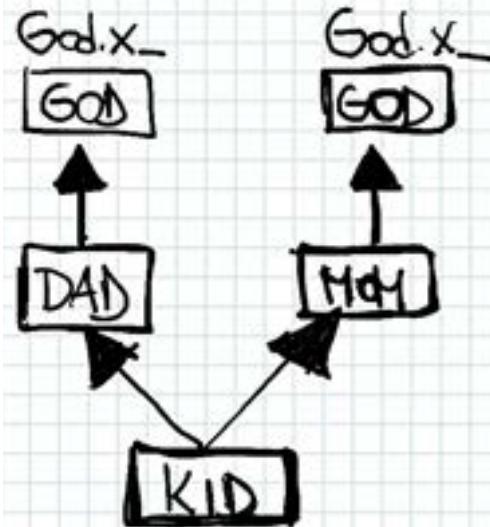
int main () {
    Kid x; // Error: with Mom and Dad having
           // independent God classes, which x_
           // should kids.print() prints?
    x.print(100);
}

```

# Class Inheritance

## Non-virtual Base Class

NON-VIRTUAL BASE



Code doesn't compile

```

whovian@phy504:~/host/CPP$ g++ class20.cpp -o class20 -std=c++20
class20.cpp: In member function ‘void Kid::print(double) const’:
class20.cpp:28:37: error: request for member ‘x_’ is ambiguous
28 |     cout << "PrintKid: " << x->x_ << endl;
          |             ^
class20.cpp:7:7: note: candidates are: ‘int God::x_’
7 |     int x_ = 2;
      |
class20.cpp:7:7: note: ‘int God::x_’

```

```

#include <string>
#include <iostream>
using namespace std;

class God {
protected:
    int x_ = 2;
public:
    void print(const double x) const {
        cout << "PrintGod: " << x->x_ << endl;
    }
};

class Mom : public virtual God {
public:
    void print(const double x) const {
        cout << "PrintMom: " << x->x_ << endl;
    }
};

class Dad : public virtual God {
public:
    void print(const double x) const {
        cout << "PrintDad: " << x->x_ << endl;
    }
};

class Kid : public Mom, public Dad {
public:
    void print(const double x) const {
        cout << "PrintKid: " << x->x_ << endl;
    }
};

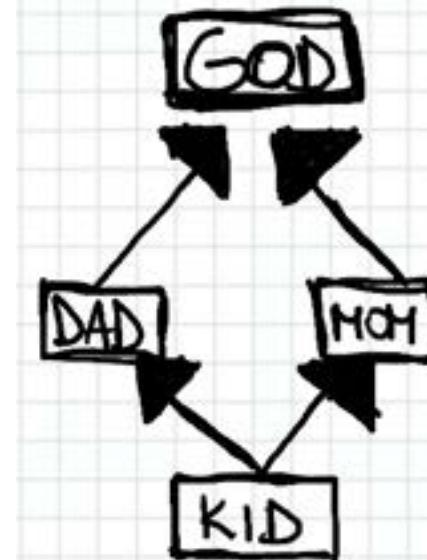
int main () {
    Kid x;
    x.print(100);
}

```

# Class Inheritance

## Virtual Base Class

VIRTUAL BASE



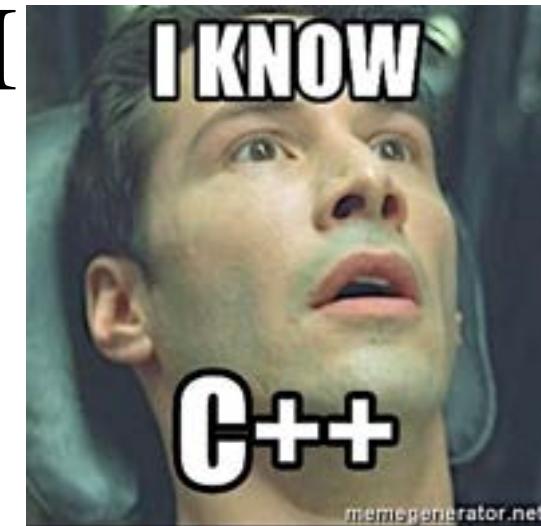
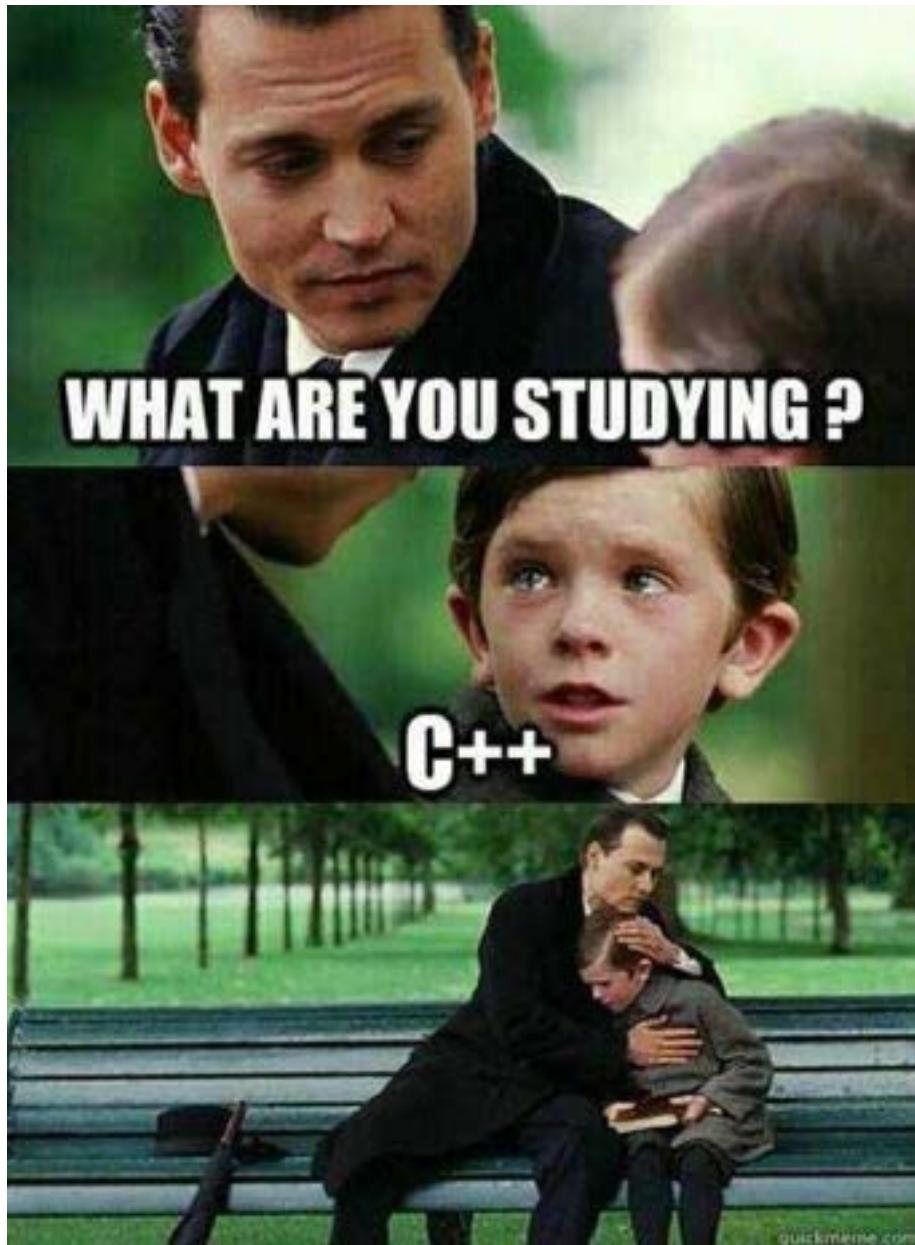
Code now runs no problem

```

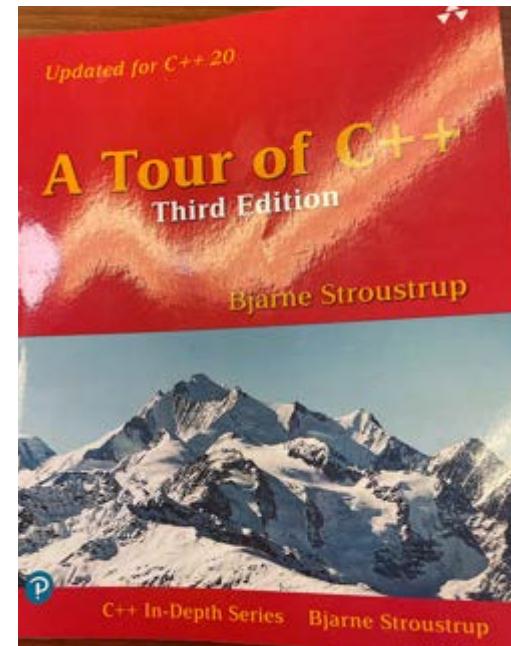
whovian@phy504:~/host/CPP$ g++ class21.cpp -o class21 -std=c++20
whovian@phy504:~/host/CPP$ ./class21
PrintKid: 98

```

# Lecture 32: C++ part VIII



## Suggested Literature



# C++ Classes Part III

```
class Base {  
public:  
    void print() {  
        // code  
    }  
};  
  
class Derived : public Base {  
public:  
    void print() {  
        // code  
    }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```



```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

void myglobal(Base* y) {
    y->print(100);
}

int main () {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);

    myglobal(y);
    // you can cast directly from new
    Base* y2 = new Derived();
    myglobal(y2);
}
```

# Class Inheritance

## Abstract **Base** Class

Abstract base class  
cannot be created, only  
dynamically casted from  
derived class

```
whovian@phy504:~/host/CPP$ g++ class18v2.cpp -o class18v2 -std=c++20
whovian@phy504:~/host/CPP$ ./class18v2
PrintDerived: 90
PrintDerived: 90
```

```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    // Error: no print2 function on Base (override)
    void print2(const double x) const override {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main() {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);
}
```

# Class Inheritance

If you are 100% sure the derived class member function must override a virtual function on Base, you can use the optional keyword **override** for additional checks

```
whovian@phy504:~/host/CPP$ g++ class18v3.cpp -o class18v3 -std=c++20
class18v3.cpp:14:8: error: 'void Derived::print2(double) const' marked 'override', but does not override
  14 |     void print2(const double x) const override {
      |             ^
class18v3.cpp: In function 'int main()':
class18v3.cpp:20:11: error: cannot declare variable 'x' to be of abstract type 'Derived'
  20 |     Derived x;
      |             ^
class18v3.cpp:10:7: note: because the following virtual functions are pure within 'Derived':
  10 | class Derived : public Base {
      |             ^
class18v3.cpp:7:16: note:     'virtual void Base::print(double) const'
   7 |     virtual void print(const double x) const = 0;
      |             ^
```

```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    // Error: wrong overloading (base version has arg)
    void print() const override {
        cout << "PrintDerived: " << 10 - this->x_ << endl;
    }
};

int main() {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);
}
```

# Class Inheritance

If you are 100% sure the derived class member function must override a virtual function on Base, you can use the optional keyword **override** for additional checks

```
whovian@phy504:~/host/CPP$ g++ class18v4.cpp -o class18v4 -std=c++20
class18v4.cpp:14:8: error: ‘void Derived::print() const’ marked ‘override’, but does not override
  14 |     void print() const override {
      |     ^
class18v4.cpp: In function ‘int main()’:
class18v4.cpp:20:11: error: cannot declare variable ‘x’ to be of abstract type ‘Derived’
  20 |     Derived x;
      |     ^
class18v4.cpp:10:7: note:   because the following virtual functions are pure within ‘Derived’:
  10 | class Derived : public Base {
      |     ^
class18v4.cpp:7:16: note:     ‘virtual void Base::print(double) const’
    7 |     virtual void print(const double x) const = 0;
      |     ^
```

```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    // Error: wrong overloading again (forgot: const)
    void print(const double x) override {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main() {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);
}
```

# Class Inheritance

If you are 100% sure the derived class member function must override a virtual function on Base, you can use the optional keyword **override** for additional checks

```
whovian@phy504:~/host/CPP$ g++ class18v5.cpp -o class18v5 -std=c++20
class18v5.cpp:14:8: error: 'void Derived::print(double)' marked 'override', but does not override
  14 |     void print(const double x) override {
      |           ^
class18v5.cpp: In function 'int main()':
class18v5.cpp:20:11: error: cannot declare variable 'x' to be of abstract type 'Derived'
  20 |     Derived x;
      |           ^
class18v5.cpp:10:7: note: because the following virtual functions are pure within 'Derived':
  10 | class Derived : public Base {
      |           ^
class18v5.cpp:7:16: note:   'virtual void Base::print(double) const'
  7 |     virtual void print(const double x) const = 0;
      |           ^
```

```
#include <string>
#include <iostream>
using namespace std;

class Base {
public:
    void print(const int x) const {
        const int tmp = x - this->x_;
        cout << "PrintBase (int): " << tmp << endl;
    }

    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintBase (double): " << tmp << endl;
    }
protected:
    int x_ = 2;
};

class Derived : public Base {
public:
    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintDerived (double): " << tmp << endl;
    }
};

int main ()
{
    Base y;
    y.print(100.1);
    y.print(5);

    Derived x;
    x.print(100.1);
    x.print(5);
}
```

# Function overloading

Member function  
overloading rules may  
surprise you.

They don't work  
automatically across scopes  
(from base to derived)

```
whovian@phy504:~/host/CPP$ g++ class28.cpp -std=c++20
whovian@phy504:~/host/CPP$ ./class28
PrintBase (double): 98.1
PrintBase (int): 3
PrintDerived (double): 98.1
PrintDerived (double): 3
```

```
#include <string>
#include <iostream>
using namespace std;

class Base {
public:
    void print(const int x) const {
        const int tmp = x - this->x_;
        cout << "PrintBase (int): " << tmp << endl;
    }

    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintBase (double): " << tmp << endl;
    }
protected:
    int x_ = 2;
};

class Derived : public Base {
public:
    using Base::print;

    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintDerived (double): " << tmp << endl;
    }
};

int main ()
{
    Base y;
    y.print(100.1);
    y.print(5);

    Derived x;
    x.print(100.1);
    x.print(5);
}
```

# Function overloading

Member function  
overloading rules: they don't  
work across scopes (from  
base to derived)

Workaround? This behavior  
can be modified with the  
**using** statement

```
whovian@phy504:~/host/CPP$ g++ class29.cpp -o class29 -std=c++20
whovian@phy504:~/host/CPP$ ./class29
PrintBase (double): 98.1
PrintBase (int): 3
PrintDerived (double): 98.1
PrintBase (int): 3
```

```

#include <iostream>
#include <spdlog/spdlog.h>
#include <fmt/core.h>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() // overloading default constructor
    {   // this ensures proper array init
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }

    double operator[](const int i) {
        return this->x_[i];
    }

    double operator()(const int i) { // safer version
        if (i < 0) {
            spdlog::critical("negative index i");
            exit(1);
        }
        else if (i > maxsz) {
            spdlog::critical(
                fmt::format(
                    "index out of bounds (max={:3d})", maxsz));
            exit(1);
        }
        return this->x_[i];
    }
};

int main () {
    Base x;
    cout << x[50] << endl;
    cout << x(200) << endl;
    cout << x(2000) << endl;
}

```

# Operators in C++

Class can have operators  
(unary and binary operators)

There are many operators that  
can be defined / overloaded

+	-	*	/	%	<sup>^</sup>	&
	~	!	=	<	>	<sup>+=</sup>
-=	<sup>+=</sup>	<sup>/=</sup>	<sup>%=</sup>	<sup>^=</sup>	<sup>&amp;=</sup>	<sup> =</sup>
<<	>>	>>=	<<=	=	<sup>!=</sup>	<sup>&lt;=</sup>
<sup>≥</sup>	<sup>&amp;&amp;</sup>	<sup>  </sup>	<sup>++</sup>	<sup>--</sup>	<sup>-&gt;*</sup>	,
<sup>&gt;</sup>	<sup>[]</sup>	<sup>0</sup>	<sup>new</sup>	<sup>new[]</sup>	<sup>delete</sup>	<sup>delete[]</sup>

```

whovian@phy504:~/host/CPP$ g++ class22.cpp -o class22 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./class22
50
200
[2023-04-07 15:58:59.955] [critical] index out of bounds (max=1000)

```

```
#include <iostream>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    Base operator++() { // prefix operator ++x
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += 1;
        }
        return *this;
    }
    Base operator++(int) { // postfix operator (x++)
        // int is a dummy argument that tells the compiler
        // this is the postfix term. Weird, but that was
        // a choice of C++ original designers. See
        // https://stackoverflow.com/a/12740593/2472169
        Base tmp = *this;
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += 1;
        }
        return tmp; //copy of the object before increment
    }
};

int main () {
    Base x;
    Base y = x++;
    cout << y[50] << endl;

    Base x2;
    Base y2 = ++x2;
    cout << y2[50] << endl;
}
```

# Operators in C++

Prefix and postfix operators are interesting (and a bit weird) to overload

Why the postfix needs an integer dummy variable?

Choice of the original C++ designers

```
whovian@phy504:~/host/CPP$ g++ class23.cpp -o class23 -std=c++20
whovian@phy504:~/host/CPP$ ./class23
50
51
```

```
#include <iostream>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    Base operator++() { // prefix operator ++
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += 1;
        }
        return *this;
    }
    // args by reference (explanation next class!)
    friend bool operator==(Base& l, Base& r);
};

// operator is a normal function (not member func)
bool operator==(Base& l, Base& r) {
    bool res = true;
    for(int i=0; i<maxsz; i++) {
        if (l.x_[i] != r.x_[i]) {
            res = false;
            break;
        }
    }
    return res;
}

int main () {
    Base x;
    ++x;
    Base y;

    bool res = (x == y);
    cout << std::boolalpha << res << endl;
}
```

# Class Inheritance

It is best for some operators to be a normal global function (not member functions).

But how a normal function can access the private data of a class? Typically, it can't. However, this limitation can be overridden with the **friend** keyword

```
whovian@phy504:~/host/CPP$ g++ class24.cpp -o class24 -std=c++20
whovian@phy504:~/host/CPP$ ./class24
false
```

```
#include <iostream>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    Base operator++() {
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += 1;
        }
        return *this;
    }
    friend Base operator+(Base& l, Base& r);
};

Base operator+(Base& l, Base& r) {
    Base res;
    for(int i=0; i<maxsz; i++)
        res.x_[i] = l.x_[i] + r.x_[i];
    return res;
}

int main () {
    Base x;
    Base y;
    Base z = x + y;
    cout << x[50] << " " << y[50]
        << " " << z[50] << endl;
}
```

# Class Inheritance

It is best for some operators to be a normal global function (not member functions).

But how a normal function can access the private data of a class? Typically, it can't.

However, this limitation can be overridden with the **friend** keyword

```
whovian@phy504:~/host/CPP$ g++ class25.cpp -o class25 -std=c++20
whovian@phy504:~/host/CPP$ ./class25
50 50 100
```

```
#include <iostream>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    // return reference = optimization
    Base& operator+=(Base& other) {
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += other.x_[i];
        }
        return *this;
    }
};

int main () {
    Base x;
    Base y;
    cout << x[50] << " " << y[50];
    x += y;
    cout << " " << x[50] << endl;
}
```

# Class Inheritance

How to decide between member or global operator?

+ - \* / should be global operators

++, +=, -=, \*=, /= should be member operators

```
whovian@phy504:~/host/CPP$ g++ class26.cpp -o class26 -std=c++20
whovian@phy504:~/host/CPP$ ./class26
50 50 100
```

```
#include <iostream>
using namespace std;
constexpr int maxsz = 10;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    friend ostream& operator<<(ostream& os, Base x);
};

ostream& operator<<(ostream& os, Base x)
{
    os << "Printing Base \n";
    for(int i=0; i<maxsz; i++)
        os << x.x_[i] << " ";
    return os;
}

int main () {
    Base x;
    cout << x << endl;
}
```

# Operators in C++

How to overload  
output stream <<  
(alternative way to  
print the class instead  
of a print() method)?

Output stream <<  
should be a global  
operator

```
whovian@phy504:~/host/CPP$ g++ class27.cpp -o class27 -std=c++20
whovian@phy504:~/host/CPP$ ./class27
Printing Base
0 1 2 3 4 5 6 7 8 9
```

```
#include <iostream>
#include <unistd.h>
using namespace std;
constexpr int maxsz = 600000000;

class Base {
    double* x_;
public:
    Base(); // Class Constructor
    ~Base(); // Class Destructor
};

Base::Base() {
    this->x_ = new double[maxsz];
}

Base::~Base() { // Class Destructor
    cout << "Destructor called" << endl;
    if(this->x_ != nullptr) {
        delete [] this->x_;
        cout << "Ptr deleted" << endl;
    }
}

int main () {
    for (int i=0; i<5; i++) {
        Base y;
        sleep(2);
    }
}
```

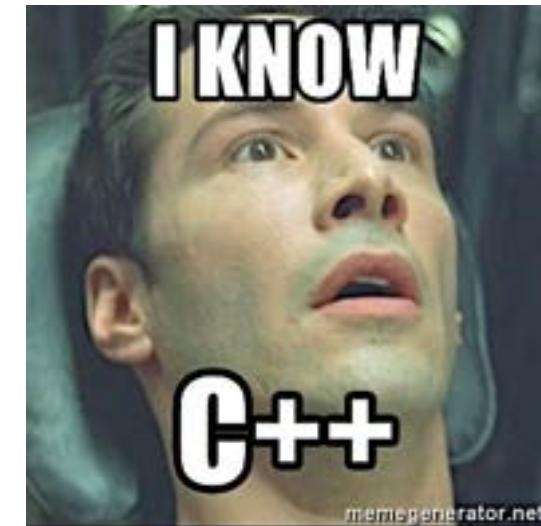
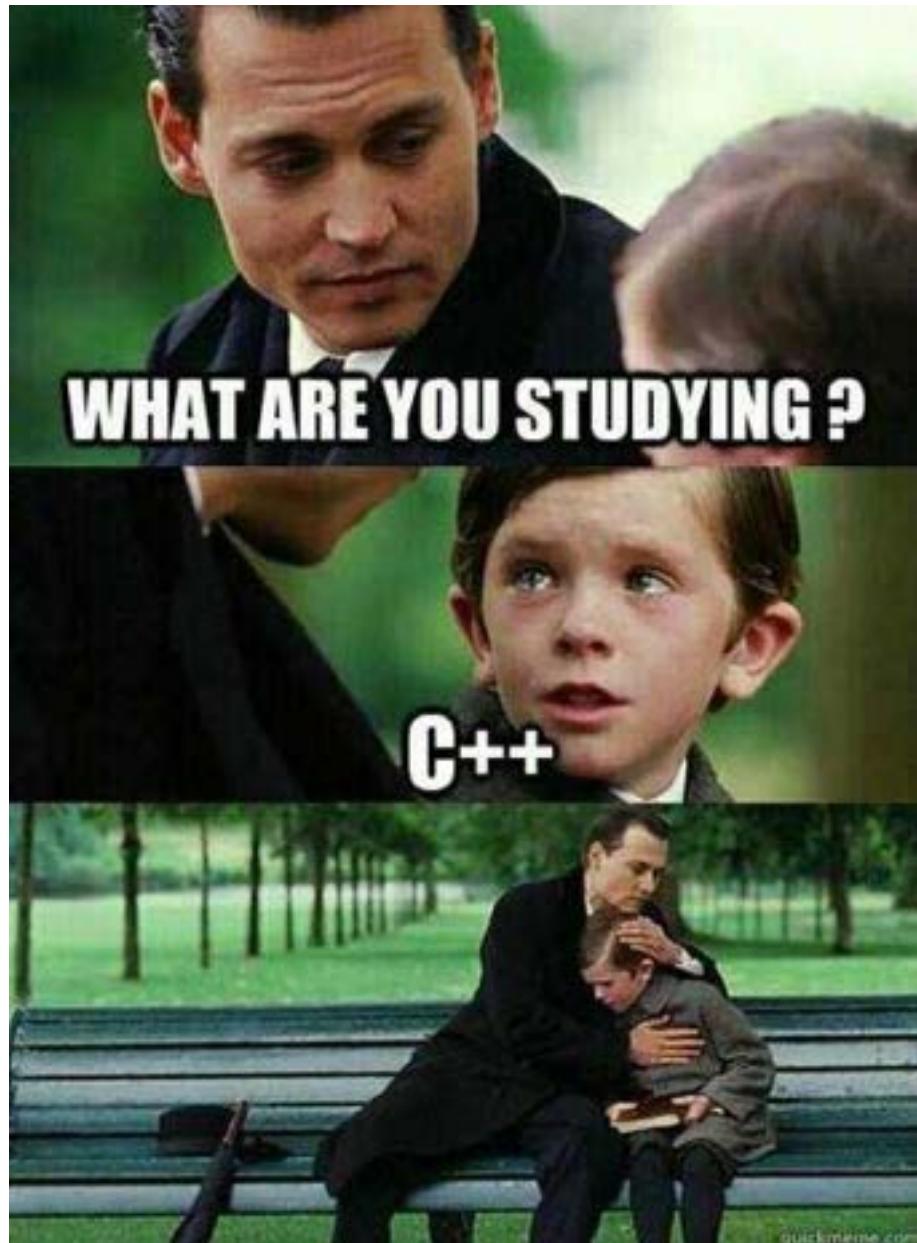
# Memory Management

If a member variable is allocated via **new** operator (C++ version of **malloc**), how can we avoid memory leaks?

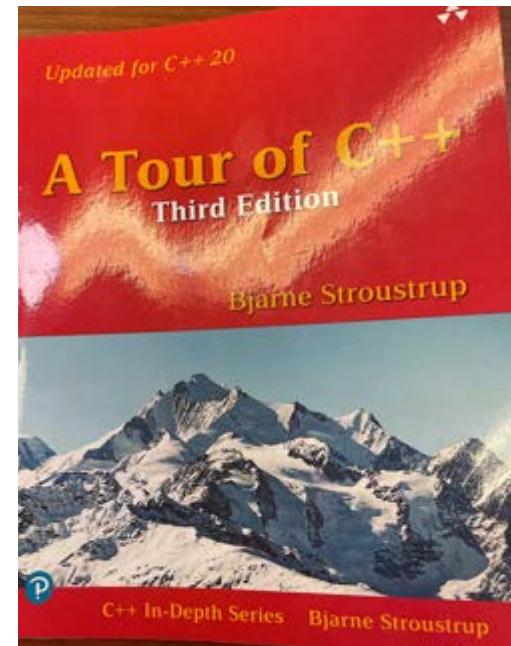
Answer: [RAII](#) (Resource Acquisition is Initialization)

```
whovian@phy504:~/host/CPP$ g++ class30.cpp -o class30 -std=c++20 -O0
whovian@phy504:~/host/CPP$ ./class30
Destructor called
Ptr deleted
```

# Lecture 33: C++ part IX



## Suggested Literature



# C++ Classes Part IV

```
class Base {  
public:  
    void print() {  
        // code  
    }  
};  
  
class Derived : public Base {  
public:  
    void print() {  
        // code  
    }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```



```
#include <iostream>
#include <unistd.h>
using namespace std;
constexpr int maxsz = 600000000;

class Base {
    double* x_;
public:
    Base(); // Class Constructor
    ~Base(); // Class Destructor
};

Base::Base() {
    this->x_ = new double[maxsz];
}

Base::~Base() { // Class Destructor
    cout << "Destructor called" << endl;
    if(this->x_ != nullptr) {
        delete [] this->x_;
        cout << "Ptr deleted" << endl;
    }
}

int main () {
    for (int i=0; i<5; i++) {
        Base y;
        sleep(2);
    }
}
```

# Memory Management

If a member variable is allocated via **new** operator (C++ version of **malloc**), how can we avoid memory leaks?

Answer: [RAII](#) (Resource Acquisition is Initialization)

```
whovian@phy504:~/host/CPP$ g++ class30.cpp -o class30 -std=c++20 -O0
whovian@phy504:~/host/CPP$ ./class30
Destructor called
Ptr deleted
```

```

#include <iostream>
#include <unistd.h>
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base(); // Class Constructor
    ~Base(); // Class Destructor
};

Base::Base() {
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++) {
        this->x_[i] = i;
    }
}

Base::~Base() { // Class Destructor
    cout << "Destructor called" << endl;
    if(this->x_ != nullptr) {
        delete [] this->x_;
        cout << "Ptr deleted" << endl;
    }
}

int main ()
{
    Base a;
    Base b(a); // copy constructor called
    a.x_[50] = -1;
    // two bugs!
    // Bug 1: a.x_[50] == b.x_[50]
    // Why? C++ copied the pointer not the data
    // Bug 2: at the end of the program, we will
    // see a double deletion of the same data array
    cout << a.x_[50] << " " << b.x_[50] << endl;
}

```

# Memory Management

Adoption of the **new** operator creates difficulties in copying class instantiations (objects)

Answer: copying the pointer is not the same as copying the underlying data

```

whovian@phy504:~/host/CPP$ g++ class31.cpp -o class31 -std=c++20 -O0
whovian@phy504:~/host/CPP$ ./class31
-1 -1
Destructor called
Ptr deleted
Destructor called
double free or corruption (!prev)
qemu: uncaught target signal 6 (Aborted) - core dumped
Aborted

```

```
#include <iostream>
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base(); // Class Default Constructor
    ~Base(); // Class Destructor
    Base(const Base& other); // Copy Constructor
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main()
{
    Base a;
    Base b(a); // copy constructor called
    a.x_[50] = -1;
    cout << a.x_[50] << " " <<
                    b.x_[50] << endl;
}
```

# Memory Management

If using **new** / **delete** operators explicitly (bad idea!), then we need to teach the class how to copy (and move) the data

How? Answer: **copy** and **move** (new concept) **constructors** (and **assignment operators**).

```
whovian@phy504:~/host/CPP$ g++ class32.cpp -o class32 -std=c++20
whovian@phy504:~/host/CPP$ ./class32
Default Constructor
Copy Constructor
-1 50
Destructor
Destructor
```

```
#include <iostream>
#include <memory> // swap
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base() { // Default Constructor
        Base();
        ~Base(); // Class Destructor
        Base(const Base& other); // Copy Constructor
        Base& operator=(Base other); // Assignment operator
        // implemented via copy
        // and swap idiom
    }

    Base::Base() {
        cout << "Default Constructor" << endl;
        this->x_ = new double[maxsz];
        for (int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }

    Base::Base(const Base& other) {
        cout << "Copy Constructor" << endl;
        this->x_ = new double[maxsz];
        for (int i=0; i<maxsz; i++)
            this->x_[i] = other.x_[i];
    }

    Base& Base::operator=(Base other) {
        // On copy-and-swap idiom, arg is passed
        // by value (it will trigger the copy constructor)
        cout << "operator = (assignment)" << endl;
        swap(this->x_, other.x_);
        return *this;
    }

    Base::~Base() {
        cout << "Destructor" << endl;
        if(this->x_ != nullptr)
            delete [] this->x_;
    }
}

int main () {
    Base a;
    Base b;
    b = a;
    a.x_[50] = -1;
    cout << a.x_[50] << " " << b.x_[50] << endl;
}
```

# Memory Management

There are nuances on defining assignment operator.

Usually adopted strategies based on the **copy constructor** ([copy-and-swap idiom](#))

```
whovian@phy504:~/host/CPP$ g++ class33.cpp -o class33 -std=c++20
whovian@phy504:~/host/CPP$ ./class33
Default Constructor
Default Constructor
Copy Constructor
operator = (assignment)
Destructor
-1 50
Destructor
Destructor
```

```
#include <iostream>
#include <memory> // swap
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base() { // Default Constructor
        ~Base(); // Class Destructor
    }
    Base(const Base& other); // Copy Constructor
    Base& operator=(Base other); // Assignment operator
        // implemented via copy
        // and swap idiom
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base& Base::operator=(Base other) {
    // On copy-and-swap idiom, arg is passed
    // by value (it will trigger the copy constructor)
    cout << "operator = (assignment)" << endl;
    swap(this->x_, other.x_);
    return *this;
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main () {
    Base a;
    Base b = a;
    a.x_[50] = -1;
    cout << a.x_[50] << " " << b.x_[50] << endl;
}
```

# Memory Management

There are nuances on defining assignment operator.

If you code instead **Base b = a;** (assignment on initialization), compiler may call the copy constructor instead

```
whovian@phy504:~/host/CPP$ g++ class34.cpp -o class34 -std=c++20
whovian@phy504:~/host/CPP$ ./class34
Default Constructor
Copy Constructor
-1 50
Destructor
Destructor
```

```
#include <iostream>
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base();
    ~Base();
    Base(const Base& other) = delete;
    Base& operator=(Base other) = delete;
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main () {
    Base a;
    Base b = a;
}
```

# Memory Management

Finally, can I prevent a class from having an instance copied

Yes – we can delete the **copy constructor** and **assignment operator**

```
whovian@phy504:~/host/CPP$ g++ class35.cpp -o class35 -std=c++20
class35.cpp: In function ‘int main()’:
class35.cpp:29:12: error: use of deleted function ‘Base::Base(const Base& )’
  29 |     Base b = a;
                 ^
class35.cpp:10:3: note: declared here
  10 |     Base(const Base& other) = delete;
                 | ^~~~~~
```

```
#include <iostream>
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base();
    ~Base();
    Base(const Base& other) = delete;
    Base& operator=(Base other) = delete;
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main () {
    Base a;
    Base b;
    b = a;
}
```

# Memory Management

Finally, can I prevent a class from having an instance copied?

Yes – we can delete the copy constructor and assignment operator

```
whovian@phy504:~/host/CPP$ g++ class36.cpp -o class36 -std=c++20
class36.cpp: In function 'int main()':
class36.cpp:30:7: error: use of deleted function 'Base& Base::operator=(Base)'
      30 |     b = a;
           |     ^
class36.cpp:11:9: note: declared here
      11 |     Base& operator=(Base other) = delete;
           |     ^~~~~~
```

# l-value references

# l-value references

```
#include <iostream>

using namespace std;

void Print(double x)
{ // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

int main()
{
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    return 0;
}
```

whovian@phy504:~/host/CPP\$ g++ ref1.cpp -o ref1 -std=c++20  
whovian@phy504:~/host/CPP\$ ./ref1  
Print: 10  
Main: 10

In C, arguments are **passed by value**.

Without the use of pointers, where the address of a variable is passed by value, there is no way in C that function arguments can modify variables defined on **main()**

# l-value references

```
#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    // passed by reference (no copy)
    // not exactly a pointer (cannot change
    // x to be address of another variable)
    // + no need to use the *x operator

    cout << "Print2: " << x << endl;
    x = 2;
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl;

    return 0;
}
```

In C++, arguments can be passed by **l-value reference**

References are not pointer

No need for **\*x** dereference operator (like in pointers)

Cannot change which “address” the function argument references

```
whovian@phy504:~/host/CPP$ g++ ref2.cpp -o ref2 -std=c++20
whovian@phy504:~/host/CPP$ ./ref2
Print: 10
Main: 10 ←
Print2: 10
Main: 2 ←
```

# l-value references

```
#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    // passed by reference (no copy)
    // not exactly a pointer (cannot change
    // x to be address of another variable)
    // + no need to use the *x operator

    cout << "Print2: " << x << endl;
    x = 2;
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl;

    return 0;
}
```

What is an **l-value** (C++03 def.)?

*Variables that makes sense at the left side of an assignment*

**x = 2;**

x is **l-value** (C++03 def.)

C++11 generalizes this definition (we won't cover)

```
whovian@phy504:~/host/CPP$ g++ ref2.cpp -o ref2 -std=c++20
whovian@phy504:~/host/CPP$ ./ref2
Print: 10
Main: 10 ←
Print2: 10
Main: 2 ←
```

# l-value references

```
#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    // passed by reference (no copy)
    // not exactly a pointer (cannot change
    // x to be address of another variable)
    // + no need to use the *x operator

    cout << "Print2: " << x << endl;
    x = 2;
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl;

    return 0;
}
```

What is an **l-value** (C++03 def.)?

*Variables that makes sense at the left side of an assignment*

**x = 2;**

**2** is **r-value** (C++03 def.)

```
whovian@phy504:~/host/CPP$ g++ ref2.cpp -o ref2 -std=c++20
whovian@phy504:~/host/CPP$ ./ref2
Print: 10
Main: 10 ←
Print2: 10
Main: 2 ←
```

# l-value references

```
#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    // passed by reference (no copy)
    // not exactly a pointer (cannot change
    // x to be address of another variable)
    // + no need to use the *x operator

    cout << "Print2: " << x << endl;
    x = 2;
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl;

    return 0;
}
```

**r-values** (C++03 def.) can't be  
*on the left side of an assignment*

**2 = x;**

This expression is never  
allowed

C++11 generalizes this  
definition (we won't cover)

```
whovian@phy504:~/host/CPP$ g++ ref2.cpp -o ref2 -std=c++20
whovian@phy504:~/host/CPP$ ./ref2
Print: 10
Main: 10 ←
Print2: 10
Main: 2 ←
```

# l-value references

```
#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    // passed by reference (no copy)
    // not exactly a pointer (cannot change
    // x to be address of another variable)
    // + no need to use the *x operator

    cout << "Print2: " << x << endl;
    x = 2;
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl;

    return 0;
}
```

**l-value** reference is a reference to an **l-value** object. In C++, arguments can be passed by **l-value reference**

Why this is relevant?

Some classes are expensive to copy (some don't allow copy)

```
whovian@phy504:~/host/CPP$ g++ ref2.cpp -o ref2 -std=c++20
whovian@phy504:~/host/CPP$ ./ref2
Print: 10
Main: 10 ←
Print2: 10
Main: 2 ←
```

```
#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    cout << "Print2: " << x << endl;
    x = 2;
}

void Print3(const double& x) {
    cout << "Print3: " << x << endl;
    // x = 2; (not allowed)
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl; Print: 10
                                         Main: 10

    Print3(x);
    cout << "Main: " << x << endl; Print2: 10
                                         Main: 2
                                         Print3: 2
                                         Main: 2
}
```

# l-value references

**l-value references** are compatible with const correctness

Copy vs reference not relevant in speed for intrinsic types  
(e.g., **int, double, char, long double, long int..**)

```
whovian@phy504:~/host/CPP$ g++ ref3.cpp -o ref3 -std=c++20
whovian@phy504:~/host/CPP$ ./ref3
Main: 10
Print2: 10
Main: 2
Print3: 2
Main: 2
```

```
#include <iostream>
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base(); // Class Default Constructor
    ~Base(); // Class Destructor
    Base(const Base& other); // Copy Constructor
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main ()
{
    Base a;
    Base b(a); // copy constructor called
    a.x_[50] = -1;
    cout << a.x_[50] << " " <<
                    b.x_[50] << endl;
}
```

# l-value references

The copy constructor needs to take arguments by **l-value** reference (const reference)

Otherwise, we would have a cyclic problem: copy constructor teaches how to copy a **Class** object, but it takes the **other** argument by value (paradox!)

```
whovian@phy504:~/host/CPP$ g++ class32.cpp -o class32 -std=c++20
whovian@phy504:~/host/CPP$ ./class32
Default Constructor
Copy Constructor
-1 50
Destructor
Destructor
```

# l-value references

```
#include <iostream>

using namespace std;

int main ()
{
    int my_super_long_name_variable = 2;
    cout << "my_super_long_name_variable = " <<
        my_super_long_name_variable << endl;

    int& x = my_super_long_name_variable;
    x = 3;

    cout << "my_super_long_name_variable = " <<
        my_super_long_name_variable << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ ref8.cpp -o ref8 -std=c++20
whovian@phy504:~/host/CPP$ ./ref8
my_super_long_name_variable = 2
my_super_long_name_variable = 3
```

**l-value** references can create multiple variables that affect the same address (similar to a pointer)

It serve as an alias (useful if original var name is cumbersome)

# l-value references

```
#include <iostream>

using namespace std;

void Print(double& x)
{
    cout << "Print: " << x << endl;
    x = 2;
}

int main()
{
    Print(10.0); // Compile Error
        // 10 is pr-value (C++11)
        // not an l-value
    return 0;
}
```

Arguments by **l-value**  
reference create a challenge:  
you can't bind it to **r-values**

```
whovian@phy504:~/host/CPP$ g++ ref4.cpp -o ref4 -std=c++20
ref4.cpp: In function ‘int main()’:
ref4.cpp:13:9: error: cannot bind non-const lvalue reference of type ‘double&’ to an rvalue of type ‘double’
  13 |     Print(10.0); // Compile Error
      |           ^
ref4.cpp:5:20: note:   initializing argument 1 of ‘void Print(double&)’
   5 | void Print(double& x)
      |           ^
```

# l-value references

```
#include <iostream>

using namespace std;

void Print(double& x)
{
    cout << "Print: " << x << endl;
    x = 2;
}

int main()          Error: r-value
{
    Print(double{10.0}); // Error

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ ref5.cpp -o ref5 -std=c++20
ref5.cpp: In function ‘int main()’:
ref5.cpp:13:9: error: cannot bind non-const lvalue reference of type ‘double&’ to an rvalue of type ‘double’
  13 |     Print(double{10.0}); // Error
      |           ^~~~~~
ref5.cpp:5:20: note:   initializing argument 1 of ‘void Print(double&)’
    5 | void Print(double& x)
      |           ^~~~~~
```

Arguments by **l-value** reference create a challenge: you can't bind it to **r-values**

# l-value references

```
#include <iostream>
#include <string>

using namespace std;

void Print(string& x)
{
    cout << "Print: " << x << endl;
}

int main()      Error: r-value
{
    Print(string("Hello")); // Error

    return 0;
}
```

Arguments by **l-value** reference create a challenge: you can't bind **r-values**

```
whovian@phy504:~/host/CPP$ g++ ref6.cpp -o ref6 -std=c++20
ref6.cpp: In function 'int main()':
ref6.cpp:13:9: error: cannot bind non-const lvalue reference of type 'std::string&' {aka 'std::__cxx11::basic_string<char>&'} to an rvalue of type 'std::string' {aka 'std::__cxx11::basic_string<char>'}
  13 |     Print(string("Hello")); // Error
      |     ^
ref6.cpp:6:20: note:   initializing argument 1 of 'void Print(std::string&)'
   6 | void Print(string& x)
      |           ^
```

# l-value references

```
#include <iostream>
using namespace std;

class Base {
public:
    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintBase: " << tmp << endl;
    }
    int x_ = 2;
};

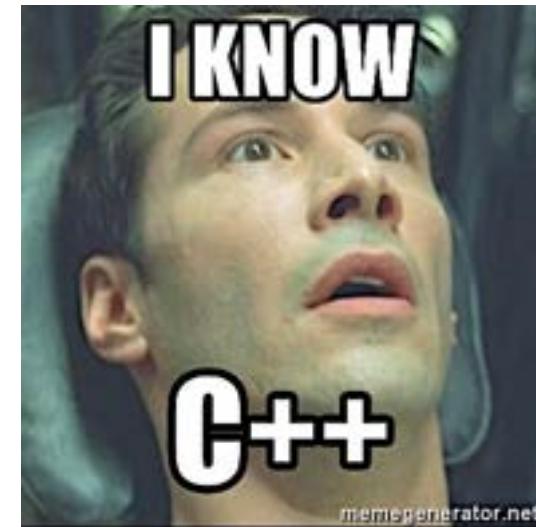
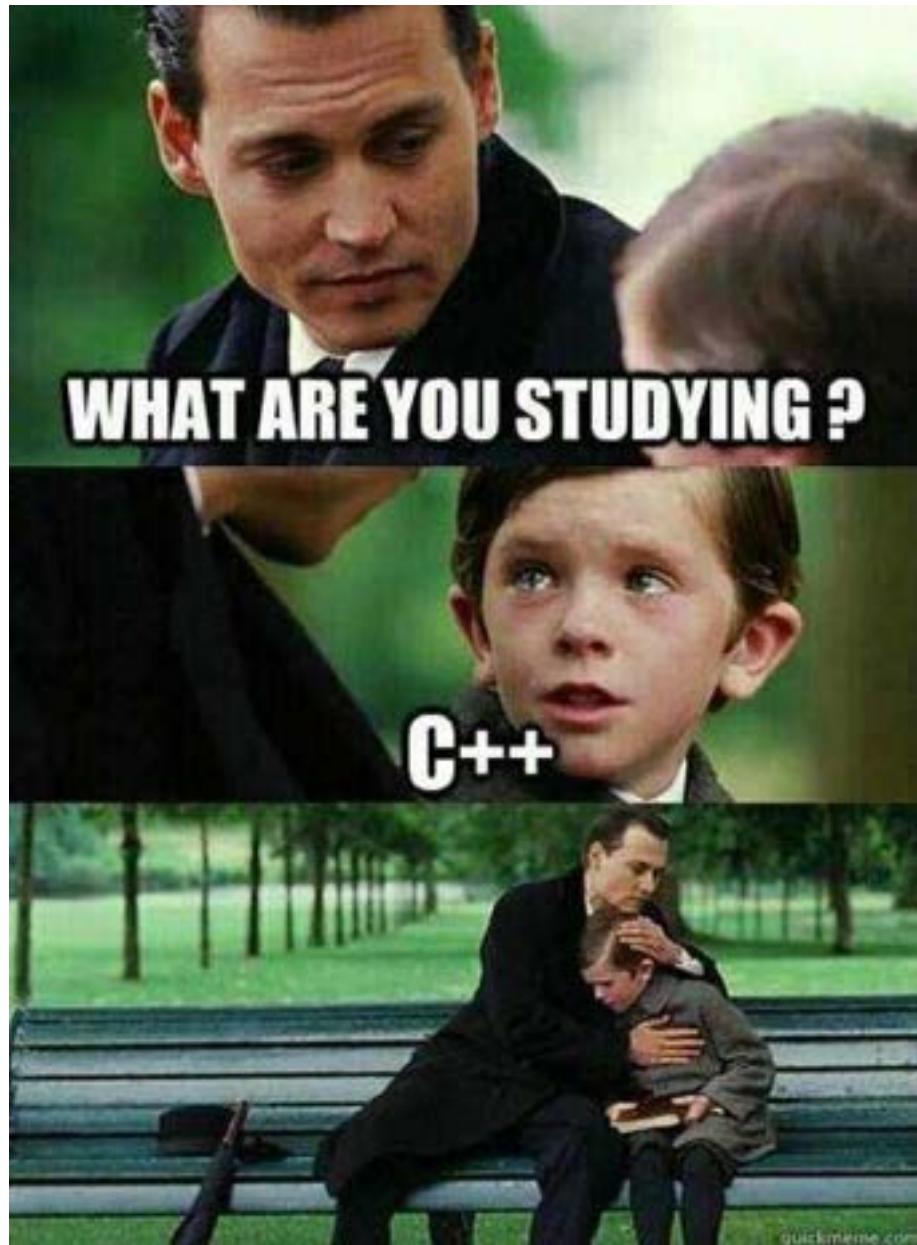
void Test(Base& a) {
    a.x_ = 2;
}

int main ()          Error: r-value
{
    Test(Base()); // error - r-value
}
```

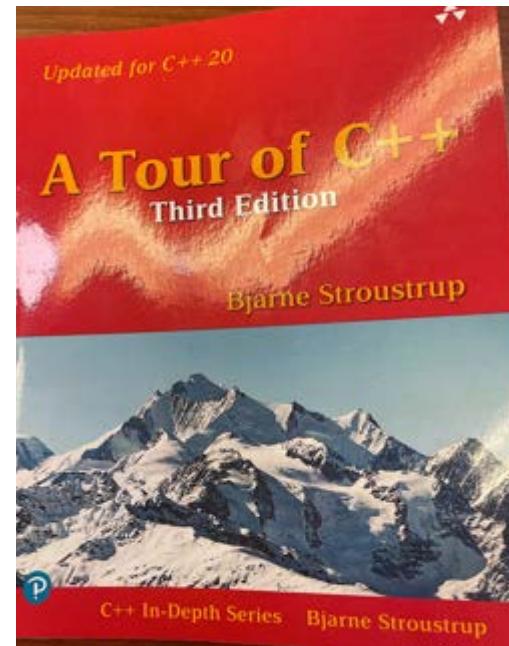
Arguments by **l-value** reference create a challenge: you can't bind **r-values**

```
whovian@phy504:~/host/CPP$ g++ ref7.cpp -o ref7 -std=c++20
ref7.cpp: In function ‘int main()’:
ref7.cpp:19:8: error: cannot bind non-const lvalue reference of type ‘Base&’ to an rvalue of type ‘Base’
  19 |     Test(Base()); // error - r-value
      |           ^
ref7.cpp:13:17: note:   initializing argument 1 of ‘void Test(Base&)’
 13 | void Test(Base& a) {
      |           ^~~~~~
```

# Lecture 34: C++ part X



## Suggested Literature



```
#include <iostream>

using namespace std;

void Print(const double& x) {
    // compiler behind the scenes creates an
    // an unnamed l-value temporary. This is
    // the so-called materialization (C++17 term)

    cout << "Print: " << x << endl;
}

void Print(const string& x) {
    cout << "Print: " << x << endl;
}

int main()
{
    Print(10);

    Print(double(15));

    Print(string("Hello World"));

    return 0;
}
```

# An interesting example

**r-values** can bind to  
**const l-value**  
**references**. How?

The **const** keyword  
prevents the l-value  
reference from being  
on the left side of any  
assignment!

```
whovian@phy504:~/host/CPP$ g++ ref9.cpp -o ref9 -std=c++20
whovian@phy504:~/host/CPP$ ./ref9
Print: 10
Print: 15
Print: Hello World
```

```
#include <iostream>

using namespace std;

void Print(const double& x) {
    // compiler behind the scenes creates an
    // an unnamed l-value temporary. This is
    // the so-called materialization (C++17 term)

    cout << "Print: " << x << endl;
}

void Print(const string& x) {
    cout << "Print: " << x << endl;
}

int main()
{
    Print(10);

    Print(double(15));

    Print(string("Hello World"));

    return 0;
}
```

# An interesting example

Compiler creates a temporary unnamed restricted l-type variable from the materialization (C++17 word) of an **r-value**

```
whovian@phy504:~/host/CPP$ g++ ref9.cpp -o ref9 -std=c++20
whovian@phy504:~/host/CPP$ ./ref9
Print: 10
Print: 15
Print: Hello World
```

```
#include <iostream>

using namespace std;

void Print(double& x)
{
    cout << "Print (l-value ref): " << x << endl;
}

void Print(double&& x)
{
    cout << "Print (r-value ref): " << x << endl;
}

int main()
{
    double x = 10;
    Print(x);

    Print(20);

    return 0;
}
```

## r-value references

**r-value references**  
bind to **r-values**

On intrinsic types, it makes no sense to use **l-value** or **r-value** references

Just pass by value and use tuples to have multiple returns

```
whovian@phy504:~/host/CPP$ g++ ref12.cpp -o ref12 -std=c++20
whovian@phy504:~/host/CPP$ ./ref12
Print (l-value ref): 10
Print (r-value ref): 20
```

```
#include <iostream>

using namespace std;

void Print(double& x)
{
    cout << "Print (l-value ref): " << x << endl;
    x = 2; // this move will propagate to main()
}

void Print(double&& x)
{
    cout << "Print (r-value ref): " << x << endl;
}

int main()
{
    double x = 10;
    Print(x);

    double y = 20;
    Print(move(y)); // data was moved(!!) from y
                    // to the r-value reference

    // Here y does not hold the data 20 anymore

    // What y holds after move? Undefined Behavior!

    // However, y is still a valid l-value variable
    y = 30; // 100% ok code!
    cout << "Print (main): " << y << endl; // 100% ok!

    return 0;
}
```

## r-value references

But on expensive types  
(classes) to be copied,  
there are good  
applications to **l-value  
references, r-value  
references** and **copy by  
value**

Example: **std::move()**

```
whovian@phy504:~/host/CPP$ g++ ref13.cpp -o ref13 -std=c++20
whovian@phy504:~/host/CPP$ ./ref13
Print (l-value ref): 10
Print (r-value ref): 20
Print (main): 30
```

```
#include <iostream>

using namespace std;

//void Print(double& x)
//{
//    cout << "Print (l-value ref): " << x << endl;
//}

void Print(double&& x) {
    cout << "Print (r-value ref): " << x << endl;
}

int main()
{
    Print(20); // ok - 20 is an r-value

    // Can r-value references deal with l-values?
    // Not automatically!
    double x = 10;
    //Print(x); // Compiler error:
    //           // error: cannot bind rvalue
    //           // reference of type 'double&&'
    //           // to lvalue of type 'double'

    Print(move(x)); // that is ok
                     // we moved the data

    // x does not hold 10 anymore
    // Again: what x holds? undefined!
    // but x is still a perfectly normal var
    x = 2; // 100% valid code
    cout << "Print (main): " << x << endl;
    return 0;
}
```

# r-value references

Can **r-value references** bind to **l-values**? No.

We need to convert **l-value** to **r-value** with **std::move()**

This (**std::move()**) has real-life consequences. The **l-value** variable does not store the data anymore

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ ref18.cpp -o ref18 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./ref18
Print (r-value ref): 20
Print (r-value ref): 10
Print (main): 2
```

```
//Hello. I want my own local copy of your Widget that I will manipulate,  
//but I don't want my changes to affect the one you have. I may or may not  
//hold onto it for later, but that's none of your business.  
void foo(Widget w);  
  
//Hello. I want to take your Widget and play with it. It may be in a  
//different state than when you gave it to me, but it'll still be yours  
//when I'm finished. Trust me!  
void foo(Widget& w);  
  
//Hello. Can I see that Widget of yours? I don't want to mess with it;  
//I just want to check something out on it. Read that one value from it,  
//or observe what state it's in. I won't touch it and I won't keep it.  
void foo(const Widget& w);  
  
//Hello. Ooh, I like that Widget you have. You're not going to use it  
//anymore, are you? Please just give it to me. Thank you! It's my  
//responsibility now, so don't worry about it anymore, m'kay?  
void foo(Widget&& w);
```

For another way of looking at it:

```
//Here, let me buy you a new car just like mine. I don't care if you wreck  
//it or give it a new paint job; you have yours and I have mine.  
void foo(Car c);  
  
//Here are the keys to my car. I understand that it may come back...  
//not quite the same... as I lent it to you, but I'm okay with that.  
void foo(Car& c);  
  
//Here are the keys to my car as long as you promise to not give it a  
//paint job or anything like that  
void foo(const Car& c);  
  
//I don't need my car anymore, so I'm signing the title over to you now.  
//Happy birthday!  
void foo(Car&& c);
```

# r-value references

When to use the  
different  
function  
overloading  
types?

[Nice explanation](#)  
[from](#)  
[StackOverflow](#)

```

#include <iostream>
#include <memory> // swap
using namespace std;
constexpr int maxsz = 1000;

class Base {
public:
    double* x_;
    // The rule of 5: Either don't declare them, or declare them all
    Base();                                // Default Constructor      (1)
    ~Base();                               // Class Destructor        (2)
    Base(const Base& other);             // Copy Constructor        (3)
    Base& operator=(Base other);          // Assignment operator     (4)
                                         // implemented via copy
                                         // and swap idiom
    Base(Base&& other);                // Move Constructor         (5)

};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base& Base::operator=(Base other) { // copy-and-swap idiom
    cout << "operator = (assignment)" << endl;
    swap(this->x_, other.x_);
    return *this;
}

Base::Base(Base&& other) {
    cout << "Move Constructor" << endl;
    this->x_ = other.x_; // you copy the pointer to the data
    other.x_ = nullptr; // you nullify the other.x_ pointer
}

int main () {
    Base a;
    Base b;
    cout << "\n";
    b = std::move(a);
    cout << "\n";
    a = Base();
    cout << "\n";
}

```

# Move constructors

How to teach a class how to move their data to another object?

## Move constructors

```

whovian@phy504:~/host/CPP$ g++ ref14.cpp -o ref14 -std=c++20
whovian@phy504:~/host/CPP$ ./ref14
Default Constructor
Default Constructor

Move Constructor ←           No copy to arg Base other
operator = (assignment)
Destructor

Default Constructor
operator = (assignment)
Destructor

Destructor
Destructor

```

# Why the rule of 5?

If you don't define the constructors, the compiler does it for you.

If you are using STL containers (standard library), this is safe – no memory leaks

*If you define any constructor (even with the **default** keyword), that will cause the compiler to stop creating other default constructors*

Example: user-defined copy constructor disable default move constructor. If you don't understand move semantics at a high level, this may be a good

# Why the rule of 5?

	default constructor	copy constructor	copy assignment	move constructor	move assignment	destructor
user declaration of	nothing	defaulted	defaulted	defaulted	defaulted	defaulted
	any constructor	undeclared	defaulted	defaulted	defaulted	defaulted
	default constructor	user declared	defaulted	defaulted	defaulted	defaulted
	copy constructor	undeclared	user declared	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)
	copy assignment	defaulted	defaulted	user declared	undeclared (fallback enabled)	undeclared (fallback enabled)
	move constructor	undeclared	deleted	deleted	user declared	undeclared (fallback disabled)
	move assignment	defaulted	deleted	deleted	undeclared (fallback disabled)	user declared
	destructor	defaulted	defaulted	defaulted	undeclared (fallback enabled)	user declared

```

#include <iostream>
#include <memory> // swap
using namespace std;
constexpr int maxsz = 1000;

class Base {
public:
    double* x_;
    Base();
    ~Base();
    Base(const Base& other);
    Base& operator=(Base other);
    // no move constructor (won't be defined by compiler)
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base& Base::operator=(Base other) { // copy-and-swap idiom
    cout << "operator = (assignment)" << endl;
    swap(this->x_, other.x_);
    return *this;
}

int main () {
    Base a;
    Base b;
    cout << "\n";
    b = std::move(a); // this will be done via copy
    cout << "\n";
    a = Base();
    cout << "\n";
}

```

# The Rule of 5

If there is no move constructor, then **std::move** is equivalent to copy (expensive)

Remember: no default move constructor with user-defined copy constructor (Rule of 5)

```

whovian@phy504:~/host/Phy504Docker/CPP$ g++ ref15.cpp -o ref15 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./ref15
Default Constructor
Default Constructor

Copy Constructor
operator = (assignment)
Destructor

Default Constructor
operator = (assignment)
Destructor

Destructor
Destructor

```

```
#include <iostream>

using namespace std;

class Base {
public:
    double x_; // just a normal double
    Base();
    ~Base() = default;
    Base(const Base& other) = default;
    // no move constructor (won't be defined by the
    // compiler). Because the user typed explicitly
    // the copy constructor (even by setting to
    // default)
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = 10;
}

int main () {
    Base a;
    Base b;
    b = std::move(a); // this will be done via copy
    a = Base();
}
```

The Rule of 5

If there is no move constructor, then **std::move** is equivalent to copy (expensive)

If this is what you want (move is an advanced feature), use the default keyword even when the copy constructor is trivial

# RVO (Return Value Optimization)

```
#include <iostream>

using namespace std;

class Base {
    int x_;
public:
    Base(int i);
    ~Base();
    Base(const Base& other);
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = other.x_;
}

Base::~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    return Base(i);
}

int main () {
    Base a = GetBase(50);
}
```

# Return Value Optimization

This is such a cool feature!

When a function returns an unnamed object, C++17 standard requires the compiler to avoid the creation of a temporary object (**copy elision**)

```
whovian@phy504:~/host/CPP$ g++ ref10.cpp -o ref10 -std=c++20
whovian@phy504:~/host/CPP$ ./ref10
Default Constructor
Destructor
```

```
#include <iostream>
using namespace std;

class Base {
    int x_;
public:
    Base(int i);
    ~Base();
    Base(const Base& other);
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = other.x_;
}

Base::~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    return Base(i);
}

int main () {
    Base a = GetBase(50); //Default Constructor
} //Destructor
```

# Return Value Optimization

**RVO** is NOT using the moving constructor

**RVO** happens even if there is no moving constructor (in this code, there isn't one because the copy constructor is used defined)

**RVO** is a compiler optimization

```
whovian@phy504:~/host/CPP$ g++ ref10.cpp -o ref10 -std=c++20
whovian@phy504:~/host/CPP$ ./ref10

```

```
#include <iostream>

using namespace std;

class Base {
    int x_;
public:
    Base(int i);
    ~Base();
    Base(const Base& other) = delete;
    Base& operator=(Base other) = delete;
    Base(Base&& other) = delete;
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    return Base(i);
}

int main () {
    Base a = GetBase(50);
}
```

whovian@phy504:~/host/Phy504Docker/CPP\$ g++ ref17.cpp -o ref17 -std=c++20  
whovian@phy504:~/host/Phy504Docker/CPP\$ ./ref17  
Default Constructor  
Destructor

# Return Value Optimization

**RVO** is NOT using the  
copying constructor  
(RVO != copy / move)

Even with deleted copying  
and move constructors, RVO  
still happens

**RVO** is a compiler  
optimization

```
#include <iostream>

using namespace std;

class Base {
    int x_;
public:
    Base(int i);
    ~Base();
    Base(const Base& other);
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = other.x_;
}

Base::~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    return Base(i);
}

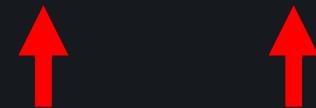
int main () {
    Base a = GetBase(50);
}
```

# Return Value Optimization

In C++<17 **RVO** could have been disabled by compiler (nice, but not a must feature)

Not possible to disable **RVO** in C++17 and beyond

```
whovian@phy504:~/host/CPP$ g++ ref10.cpp -o ref10 -std=c++14 -fno-elide-constructors  
whovian@phy504:~/host/CPP$ ./ref10
```



```
#include <iostream>

using namespace std;

class Base {
public:
    int x_;
    Base(int i);
    ~Base();
    Base(const Base& other);
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = other.x_;
}

Base::~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    Base a(i);
    cout << a.x_ << endl;
    return a;
}

int main () {
    Base b = GetBase(50);
}
```

# Return Value Optimization

In **GetBase()**, the var is named  
and RVO works on unnamed  
values

Wouldn't it be nice if the compiler could apply RVO even with named variables?

It did! Somehow the entire code happened without a copy!

```
whovian@phy504:~/host/CPP$ g++ ref11.cpp -o ref11 -std=c++20
whovian@phy504:~/host/CPP$ ./ref11
Default Constructor
50
Destructor
```

```
#include <iostream>

using namespace std;

class Base {
public:
    int x_;
    Base(int i);
    ~Base();
    Base(const Base& other);
};

Base::Base(int i) :
    x_(i) {
    cout << "Default Constructor" << endl;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = other.x_;
}

Base::~Base() {
    cout << "Destructor" << endl;
}

Base GetBase(int i) {
    Base a(i);
    cout << a.x_ << endl;
    return a;
}

int main () {
    Base b = GetBase(50);
}
```

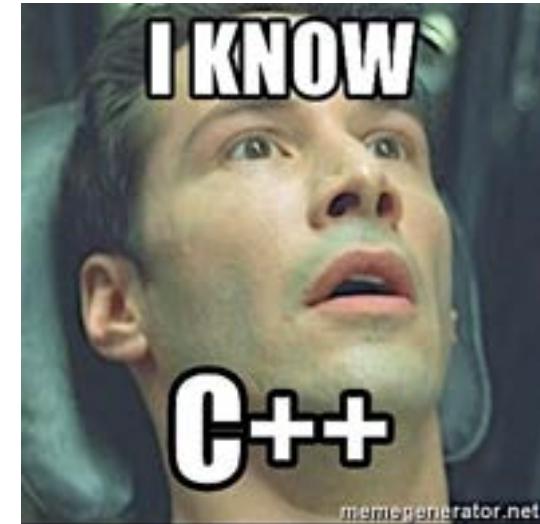
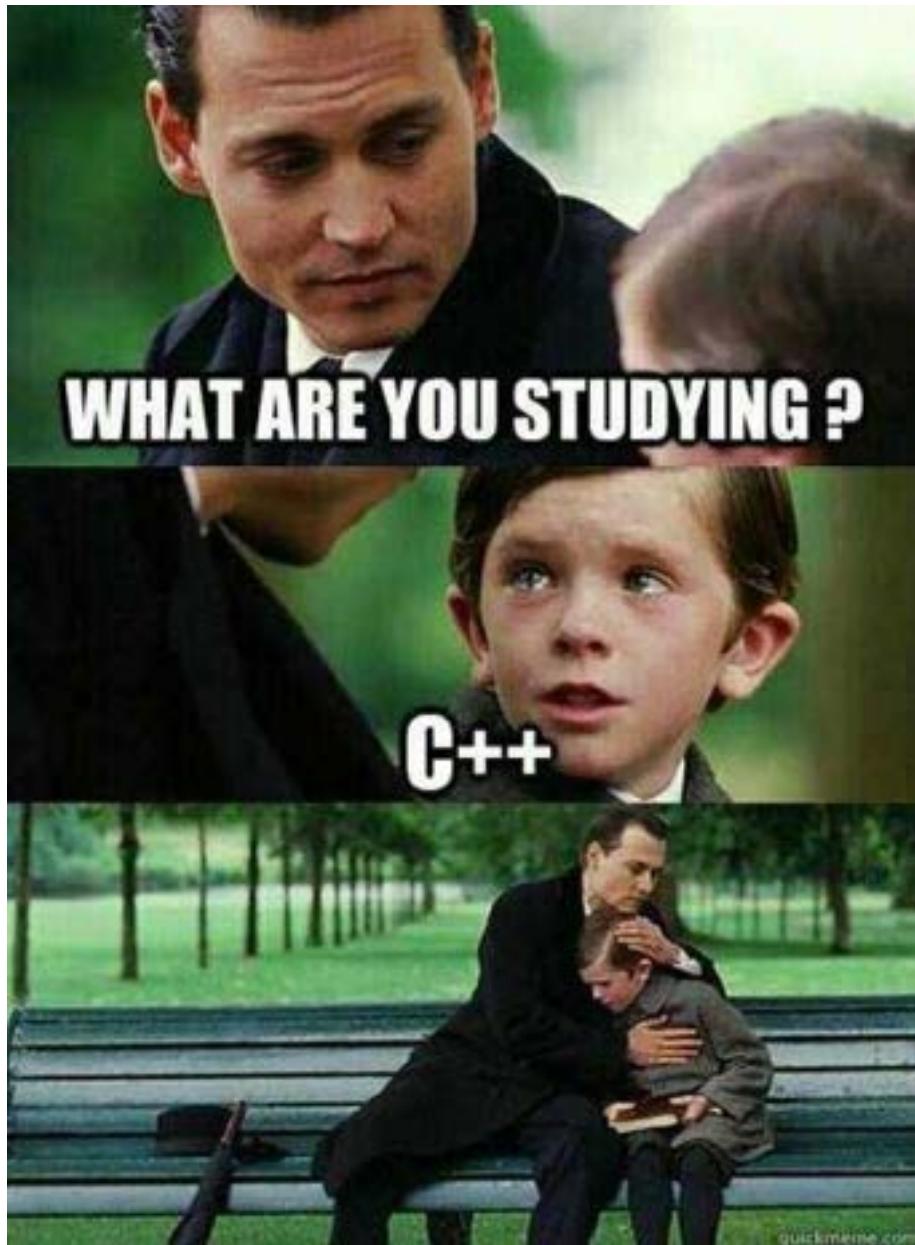
# RVO vs NRVO

This is named NRVO (C++17):  
Named Return Value Optimizations

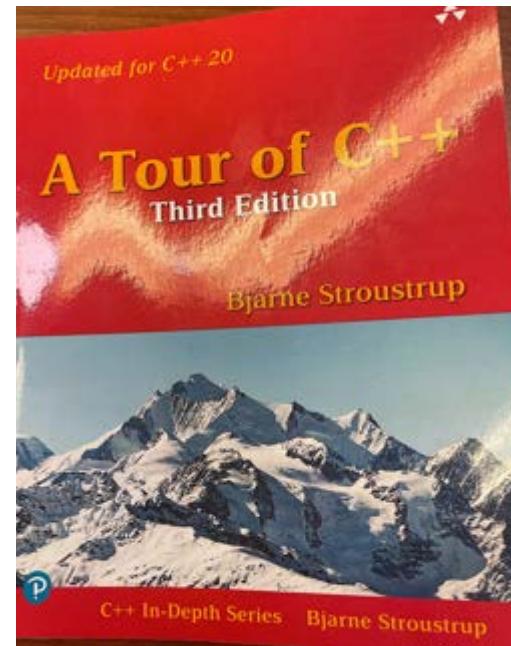
RVOs are mandatory to all compilers by the standards and cannot be turned off via compiler flags. NRVOs are just suggested and can be turned off

```
whovian@phy504:~/host/CPP$ g++ ref11.cpp -o ref11 -std=c++20
whovian@phy504:~/host/CPP$ ./ref11
Default Constructor
50
Destructor
```

# Lecture 35: C++ part XI



## Suggested Literature



```
#include <iostream>
using namespace std;

int main () {
    // in C++11, lambdas were a bit limited
    // no auto keyword on lambda's arguments
    auto pf = [] (int x, int y) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };
    pf(2, 3);

    // C++>=14, that ceased to be a problem
    auto pf2 = [] (auto x, auto y) {
        cout << x << " + " << y <<
            " = " << x+y << endl;
    };
    pf2(12.0, 20.2);

    // in C++11, lambdas was not compatible
    // with default arguments. In C++>=14,
    // they are compatible with default args

    // but default args with auto doesn't work
    auto pf3 = [] (auto x, double y = 4.0) {
        cout << x << " + " << y <<
            " = " << x+y << endl; whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda1.cpp -o lambda1 -std=c++20
    };
    pf3(1.0, 3.1);
    pf3(1.0);
}
```

# C++ Lambdas

Lambdas are anonymous functions

They are super useful in STL algorithms

They are great for readability of code

Think of them as holding small "disposable" one-time-use utility code

```
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda1
2 - 3 = -1
12 + 20.2 = 32.2
1 + 3.1 = 4.1
1 + 4 = 5
```

```

#include <iostream>
using namespace std;

int main () {
    int y = 10;

    // capture by value
    auto pf = [y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
        // y is also read-only
        // y = 10; doesn't compile
    };
    pf(20);

    // capture by l-value reference
    auto pf2 = [&y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
        y = 3;
    };
    pf2(30);
    cout << "y = " << y << endl;
}

```

# C++ Lambdas

```

whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda2.cpp -o lambda2 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda2
20 - 10 = 10
30 - 10 = 20
y = 3

```

Lambdas can “capture” existing objects in their scope (either by value or l-value reference)

Capture by value makes the variable read-only

You can override this behavior with the **mutable** keyword

```

#include <iostream>
using namespace std;

int main () {
    int y = 10;           13 |     y = 10; // doesn't compile
    // capture by value
    auto pf = [y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
        // y is also read-only
        y = 10; // doesn't compile
    };
    pf(20);

    // capture by l-value reference
    auto pf2 = [&y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
        y = 3;
    };
    pf2(30);
    cout << "y = " << y << endl;
}

```

# C++ Lambdas

```

whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda2v2.cpp -o lambda2v2 -std=c++20
lambda2v2.cpp: In lambda function:
lambda2v2.cpp:13:7: error: assignment of read-only variable 'y'

```

Lambdas can “capture” existing objects in their scope (either by value or l-value reference)

Capture by value makes the variable read-only

You can override this behavior with the **mutable** keyword

```
#include <iostream>
using namespace std;

int main ()
{
    int y = 10;

    // capture by value
    auto pf = [y](int x) mutable {
        cout << x << " - " << y <<
            " = " << x-y << endl;
        // y is also read-only
        y = 200; // 100% ok
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };
    pf(20);

    // capture by l-value reference
    auto pf2 = [&y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
        y = 3;
    };
    pf2(30);
    cout << "y = " << y << endl;
}
```

# C++ Lambdas

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda2v3.cpp -o lambda2v3 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda2v3
20 - 10 = 10
20 - 200 = -180
30 - 10 = 20
y = 3
```

Lambdas can “capture” existing objects in their scope (either by value or l-value reference)

Capture by value makes the variable read-only

You can override this behavior with the **mutable** keyword

# C++ Lambdas

```
#include <iostream>
using namespace std;

int main () {
    int y = 10;

    // capture by l-value reference
    auto pf = [&y](int x) {
        cout << x << " - " << y <<
        " = " << x-y << endl;
    };

    pf(30);

    // Be very careful with capture
    // by l-value reference
    y = 0; // affects the next call
    pf(30);
}
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda2v4.cpp -o lambda2v4 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda2v4
```

```
30 - 10 = 20
```

```
30 - 0 = 30
```

Be careful with capture by  
l-value reference

Any change to the “global”  
variable on the scope up to  
the point you call the  
function affect the local  
variable

# C++ Lambdas

```
whovian@phy504:~/host/CPP$ g++ lambda2v4v2.cpp -o lambda2v4v2 -std=c++20
whovian@phy504:~/host/CPP$ ./lambda2v4v2
30 - 10 = 20
30 - 10 = 20
```

```
#include <iostream>
using namespace std;

int main ()
{
    int y = 10;

    auto pf = [y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };

    pf(30);
    y = 0;
    pf(30);
}
```

*What about pass by value?*

Completely different behavior!

copy happens at the line where the lambda is defined!

```
#include <iostream>
using namespace std;

int main () {
    int y = 10;
    int z = 20;
    // capture by value
    auto pf = [=](int x) {
        cout << x << " - " << y <<
        " + " << z << " = " <<
        x-y+z << endl;
        // y is also read-only
        // y = 10; doesn't compile
    };
    pf(20);

    // capture by l-value reference
    auto pf2 = [&](int x) {
        cout << x << " - " << y <<
        " = " << x-y << endl;
        y = 3;
        z = 2;
    };
    pf2(30);
    cout << "y = " << y << endl;
    cout << "z = " << z << endl;
}
```

# C++ Lambdas

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda3.cpp -o lambda3 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda3
20 - 10 + 20 = 30
30 - 10 = 20
y = 3
z = 2
```

Although not advisable, lambdas can capture all the objects it sees in the scope, either by value **[=]** or by l-value reference **[&]**

```

#include <iostream>
using namespace std;

int main ()
{
    int y = 10;
    int z = 20;

    // capture by value      z = 2
    auto pf = [=](int x) mutable {
        cout << x << " - " << y <<
            " + " << z << " = " <<
            x-y+z << endl;
    y = 110;
    z = 200;
    cout << x << " - " << y <<
        " + " << z << " = " <<
        x-y+z << endl;
};

pf(20);

// capture by l-value reference
auto pf2 = [&](int x) {
    cout << x << " - " << y <<
        " = " << x-y << endl;
    y = 3;
    z = 2;
};
pf2(30);
cout << "y = " << y << endl;
cout << "z = " << z << endl;
}

```

# C++ Lambdas

```

whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda3v2.cpp -o lambda3v2 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda3v2
20 - 10 + 20 = 30
20 - 110 + 200 = 110
30 - 10 = 20
y = 3

```

Although not advisable, lambdas can capture all the objects it sees in the scope, either by value [=] or l-value references [&]

[=] + **mutable** implies variables are not read-only (same behavior as when a single capture happened)

```

#include <iostream>
using namespace std;

int factory(int i) {
    return i * 10;
}

int main () {
    int z = 10;

    auto pf = [y = z+2](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };
    pf(20);

    auto pf2 = [y = factory(2)](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };
    pf2(20);
}

```

# C++ Lambdas

C++14 introduced the **generalized lambda capture**. It allows you to:

1. specify the name of a data member in the closure class generated from the lambda
2. specify an expression initializing that data member.

```
#include <iostream>
using namespace std;

int main ()
{
    int z = 10;

    // z = z * 10 defines a local variable
    // z with value equal to the captured
    // "global" z times 10 (local z = 100)
    auto pf3 = [&r = z, z = z * 10](int x) {
        cout << "r: " << r << endl;
        ++r; // "global" captured z is now 11
        cout << "r: " << r << endl;
        cout << "z: " << z << endl;
        cout << endl;
        return r + x + z;
};

auto res = pf3(20);
cout << "pf3: " << pf3(20) << endl;
cout << endl;
// every call of pf3 changes the global z,
// which in turns change the next call of
// local z (terrible design - be careful)
auto res2 = pf3(20);
cout << "pf3: " << pf3(20) << endl;
cout << endl;
}
```

# C++ Lambdas

C++14 introduced  
the **generalized lambda  
capture**.

Be careful, with terrible  
design you can make  
your code hard to  
understand (even with  
simple calls to lambda  
functions)

```

#include <iostream>
using namespace std;

int main ()
{
    int z = 10;

    // z = z * 10 defines a local variable
    // z with value equal to the captured
    // "global" z times 10 (local z = 100)
    auto pf3 = [&r = z, z = z * 10](int x) {
        cout << "r: " << r << endl;
        ++r; // "global" captured z is now 11
        cout << "r: " << r << endl;
        cout << "z: " << z << endl;
        cout << endl;
        return r + x + z;
    };

    auto res = pf3(20);
    cout << "pf3: " << pf3(20) << endl;
    cout << endl;
    // every call of pf3 changes the global z,
    // which in turns change the next call of
    // local z (terrible design - be careful)
    auto res2 = pf3(20);
    cout << "pf3: " << pf3(20) << endl;
    cout << endl;
}

```

# C++ Lambdas

```

whovian@phy504:~/host/CPP$ ./lambda5
r: 10
r: 11
z: 100
pf3: r: 11
r: 12
z: 100
132
r: 12
r: 13
z: 100
pf3: r: 13
r: 14
z: 100
134

```

# C++ Lambdas

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda6.cpp -o lambda6 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda6
200
```

```
#include <iostream>
using namespace std;

auto get_my_lambda(int y)
{
    return [y](int x){ return x * y; };
}

int main ()
{
    auto pf = get_my_lambda(10);
    cout << pf(20) << endl;
}
```

C++14: lambda functions can be returned from normal functions

Use the **auto** keyword to specify the types

```
#include <iostream>
using namespace std;

class MyClass {
    int x_ = 0;
public:
    void f() {
        // capture this pointer by value, x is reference
        auto pf = [this](){
            this->x_++;
        };

        pf();
        cout << this->x_ << endl;
        pf();
        cout << this->x_ << endl;

        // capture *this by value, x is value
        // need mutable word to not be read-only
        auto pf2 = [*this](){
            this->x_++;
        };
        this->x_ = 0;
        pf2();
        cout << this->x_ << endl;
        pf2();
        cout << this->x_ << endl;
    }
};

int main() {
    MyClass a;
    a.f();
}
```

# C++ Lambdas

Capture **this** pointer vs  
**\*this**

First case: copy the pointer  
**this**, lambda can change  
the state of all variables in  
the surrounding class

Second case (**\*this**): copy  
the object. Lambda cannot  
change the state of the  
surrounding class

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda7.cpp -o lambda7 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda7
1
2
0
0
```

```
#include <iostream>
using namespace std;

class MyClass
{
public:
    int value {123};

    auto get_value_copy()
    {
        // Capture *this by value is C++17
        return [*this] { return value; };
    }

    auto get_value_ref()
    {
        return [this] { return value; };
    }
};

int main()
{
    MyClass a;

    auto value_copy = a.get_value_copy();
    auto value_ref = a.get_value_ref();

    a.value = 321;

    cout << "by value: " << value_copy() << endl;

    cout << "by reference: " << value_ref() <<
        endl; // (dangerous - hard to keep track)
}
```

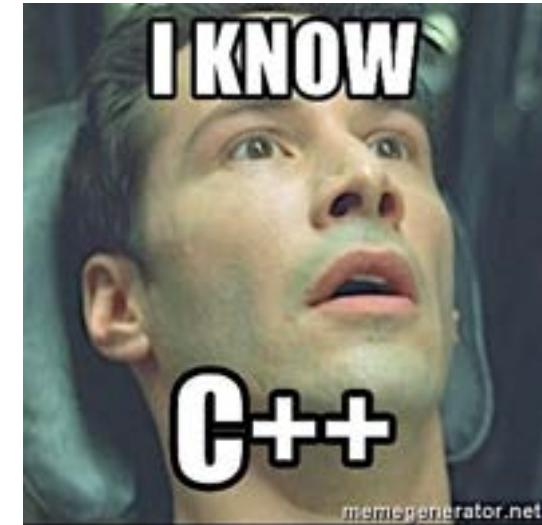
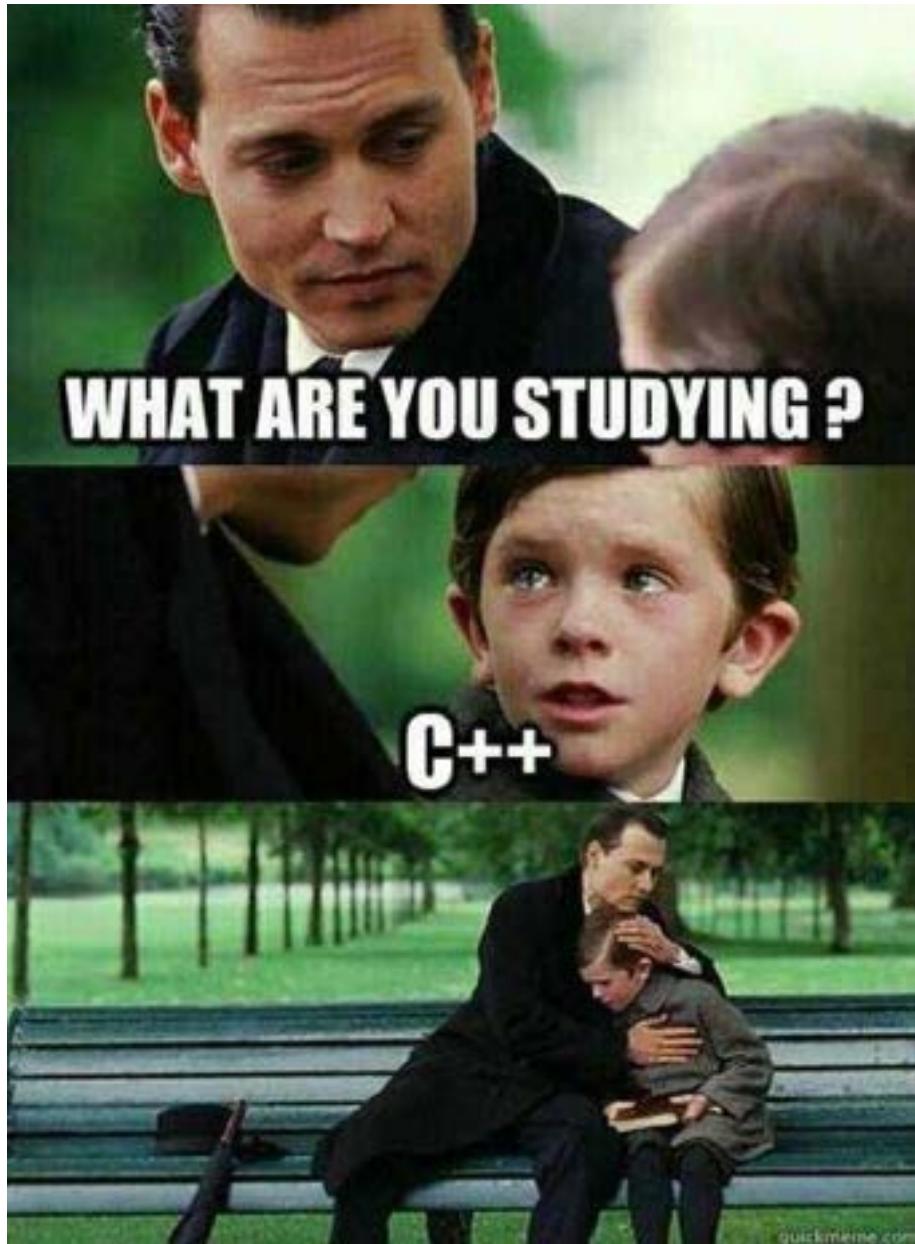
# C++ Lambdas

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda8.cpp -o lambda8 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda8
by value: 123
by reference: 321
```

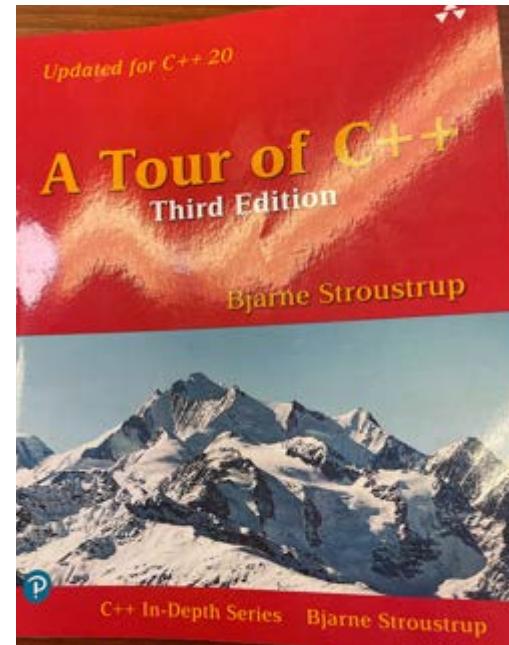
Capture **this** pointer vs  
**\*this**

Careful when capturing by reference, especially if you return the lambda from a member function

# Lecture 36: C++ part XII



## Suggested Literature



```

#include <iostream>
using namespace std;

int main () {
    int y = 10;

    // capture by l-value reference
    auto pf = [&y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };

    pf(30);

    // Be very careful with capture
    // by l-value reference
    y = 0; // affects the next call
    pf(30);
}

```

# C++ Lambdas

```

whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda2v4.cpp -o lambda2v4 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda2v4
30 - 10 = 20
30 - 0 = 30

```

*Review from last class:* Be careful with capture by l-value reference

Any change to the “global” variable on the scope up to the point you call the function affect the local variable

*What about pass by value?*

# C++ Lambdas

```
whovian@phy504:~/host/CPP$ g++ lambda2v4v2.cpp -o lambda2v4v2 -std=c++20
whovian@phy504:~/host/CPP$ ./lambda2v4v2
30 - 10 = 20
30 - 10 = 20
```

```
#include <iostream>
using namespace std;

int main ()
{
    int y = 10;

    auto pf = [y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };

    pf(30);
    y = 0;
    pf(30);
}
```

*What about pass by value?*

Completely different  
behavior!

copy happens at the line  
where the lambda is  
defined!

```
#include <iostream>
using namespace std;

int main ()
{
    int z = 10;

    // z = z * 10 defines a local variable
    // z with value equal to the captured
    // "global" z times 10 (local z = 100)
    auto pf3 = [&r = z, z = z * 10](int x) {
        cout << "r: " << r << endl;
        ++r; // "global" captured z is now 11
        cout << "r: " << r << endl;
        cout << "z: " << z << endl;
        cout << endl;
        return r + x + z;
    };

    auto res = pf3(20);
    cout << "pf3: " << pf3(20) << endl;
    cout << endl;
    // every call of pf3 changes the global z,
    // which in turns change the next call of
    // local z (terrible design - be careful)
    auto res2 = pf3(20);
    cout << "pf3: " << pf3(20) << endl;
    cout << endl;
}
```

# C++ Lambdas

*What about pass by value? Completely different behavior!*

```
whovian@phy504:~/host/CPP$ ./lambda5
r: 10
r: 11
z: 100

pf3: r: 11
r: 12
z: 100

132

r: 12
r: 13
z: 100

pf3: r: 13
r: 14
z: 100

134
```

```
#include <random>
#include <algorithm>
#include <fmt/core.h>
#include <chrono>

constexpr int sz = 5000000;

int main () {
    std::random_device rd; // a uniformly-distributed
                           // integer random number generator
                           // PS: slow & may have low entropy
                           // don't use to generate lots of
                           // random numbers

    // I will use a call of rd to seed the more robust and
    // faster Mersenne Twister integer random number gen
    std::mt19937 mt(rd()); // Mersenne Twister 32-bit gen

    // Creating a real distribution from 0 to sz
    // How to call the random number? Via operator()
    // PS: it requires mt in the arg of the operator
    std::uniform_real_distribution<double> dist(0, sz);

    std::vector<double> myvec(sz); // next class will
                                   // be about vectors

    for (auto& x : myvec) // by l-value reference
        x = dist(mt);

    auto t0 = std::chrono::high_resolution_clock::now();

    std::sort(myvec.begin(), myvec.end()); // STL algorithm

    auto t1 = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> fs = t1 - t0;

    fmt::print("Time to sort {} elements: {}\n", sz, fs.count());
}
```

# C++ Algorithms

```
whovian@phy504:~/host/CPP$ g++ algo1.cpp -o algo1 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./algo1
Time to sort 5000000 elements: 15.851820493
```

STL has a robust collection of algorithms

Algorithms can run in parallel (C++17)

In this example, we also used C++ STL random number generator and chrono libraries

```
#include <random>
#include <algorithm>
#include <fmt/core.h>
#include <chrono>
#include <execution>
constexpr int sz = 5000000;
int main () {
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, sz);
    std::vector<double> myvec(sz);

    for (auto& x : myvec)
        x = dist(mt);

    auto t0 = std::chrono::high_resolution_clock::now();

    // Massive ~300% speed on my simple Mac M1
    // I gave 4 cores to Docker Virtual Machine

    // possible std::execution policies
    // (1) std::execution::par
    // (2) std::execution::par_unseq
    // (3) std::execution::seq    // not parallel, sequenced
    // (4) std::execution::unseq // not parallel unsequenced
    std::sort(std::execution::par_unseq,
              myvec.begin(), myvec.end()); // STL algorithm

    auto t1 = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> fs = t1 - t0;
    fmt::print("Time to sort {} elements: {}\n", sz, fs.count());
}
```

# C++ Algorithms

Algorithms can run in parallel (C++17)

GCC uses [intel threading building blocks](#) under the hood for parallel execution

Performance not optimal in my M1 (4 cores to Docker)

```
#include <random>
#include <algorithm>
#include <fmt/core.h>
#include <chrono>
#include <execution>
constexpr int sz = 15'000'000; // C++17 ' (better display)

int main () {
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, sz);
    std::vector<double> v(sz);
    for (auto& x : v) // by l-value reference
        x = dist(mt);

    auto pf = [] (double& n) {
        n++;
    };

    { // just creating a local stack (so I can repeat names
        auto t0 = std::chrono::high_resolution_clock::now();

        std::for_each(v.begin(), v.end(), pf); // STL algorithm

        auto t1 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> fs = t1 - t0;
        fmt::print("Time to apply std::for_each (sum 1)\n"
                  "on {} elements: {}\n", sz, fs.count());
    }
    { // just creating a local stack (so I can repeat names
        auto t0 = std::chrono::high_resolution_clock::now();

        std::for_each(std::execution::par_unseq,
                     v.begin(), v.end(), pf); // STL algorithm

        auto t1 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> fs = t1 - t0;
        fmt::print("Time to apply std::for_each (sum 1)\n"
                  "on {} elements: {}\n", sz, fs.count());
    }
}
```

# C++ Algorithms

Algorithms can run in parallel (C++17)

Many algorithms are compatible with lambda functions

Arguments may be taken by l-value references  
(mandatory on **for\_each**)

```
whovian@phy504:~/host/CPP$ g++ algo3.cpp -o algo3 -std=c++20 -lfmt -ltbb
whovian@phy504:~/host/CPP$ ./algo3
Time to apply std::for_each (sum 1) on 15000000 elements: 1.761067834
Time to apply std::for_each (sum 1) on 15000000 elements: 0.304582292
```

```

#include <random>
#include <algorithm>
#include <fmt/core.h>
#include <chrono>
constexpr int sz = 15'000'000; // C++17 ' (better display)

int main () {
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, 1); // U(0,1)
    std::vector<double> v(sz);
    for (auto& x : v) // by l-value reference
        x = dist(mt);

    // can't run in parallel (otherwise race condition)
    class Sum
    {
    public:
        void operator()(double n)
        {
            sum += n;
        }
        double sum {0};
    };
    auto t0 = std::chrono::high_resolution_clock::now();

    // Return Value: input function. Default behavior
    // of STL algorithm (perfect for functors)
    Sum s = std::for_each(v.begin(), v.end(), Sum());
    auto t1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> fs = t1 - t0;

    fmt::print("Time to apply std::for_each (sum functor)"
              " on {} elements: {}.\n", sz, fs.count());
    fmt::print("Value of the sum: {}\n", s.sum());
}

```

# C++ Algorithms

Many algorithms are compatible w/ **functors**

**Functors** are C++ classes with **operator()**.

**Functors** can hold variables as members

Algorithms like **for\_each** return the function itself (perfect / designed for functors)

```

whovian@phy504:~/host/CPP$ g++ algo4.cpp -o algo4 -std=c++20 -lfmt -ltbb
whovian@phy504:~/host/CPP$ ./algo4
Time to apply std::for_each (sum functor) on 150000000 elements: 1.210009834
Value of the sum: 7499073.257257113

```

```
#include <random>
#include <algorithm>
#include <fmt/core.h>
#include <chrono>
#include <execution>
constexpr int sz = 15'000'000; // C++17 ' (better display

int main () {
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, sz);
    std::vector<double> v(sz);
    for (auto& x : v) // by l-value reference
        x = dist(mt);

    auto pf = [] (double& n) {
        n++;
    };
{
    auto t0 = std::chrono::high_resolution_clock::now();

    std::for_each(std::execution::par_unseq,
                 v.begin(), v.end(), pf); // STL Algorithm

    auto t1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> fs = t1 - t0;
    fmt::print("Time to apply std::for_each"
              " on {0} elements: {1}\n", sz, fs.count());
}
{
    auto t0 = std::chrono::high_resolution_clock::now();

    for (auto& x : v) // by l-value reference
        x++;

    auto t1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> fs = t1 - t0;
    fmt::print("Time to apply python style loop"
              " on {0} elements: {1}\n", sz, fs.count());
}
```

# C++ Algorithms

The elephant in the room

What is the difference  
between using STL  
algorithm **for\_each** and  
python style for loop  
(or any for loop)?

**for\_each** is simpler to  
parallelize

```
whovian@phy504:~/host/CPP$ g++ algo3v2.cpp -o algo3v2 -std=c++20 -lfmt -ltbb
whovian@phy504:~/host/CPP$ ./algo3v2
Time to apply std::for_each on 15000000 elements: 0.249909792
Time to apply python style loop on 15000000 elements: 1.300911001
```

```
#include <random>
#include <algorithm>
#include <fmt/core.h>
#include <chrono>
#include <execution>
#include <cmath>
constexpr int sz = 15'000'000; // C++17 ' (better display)

int main () {
    std::random_device rd; std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, sz);
    std::vector<double> v(sz);
    for (auto& x : v) // by l-value reference
        x = dist(mt);

    auto pf = [] (double n) {
        return std::sqrt(std::sin(n)*std::cos(n));
    };
    {
        auto t0 = std::chrono::high_resolution_clock::now();

        std::vector<double> vout(sz);
        std::transform(v.begin(), v.end(), vout.begin(), pf);

        auto t1 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> fs = t1 - t0;
        fmt::print("Time to apply std::transform"
                  " on {} elements: {}\n", sz, fs.count());
    }
    {
        auto t0 = std::chrono::high_resolution_clock::now();

        std::vector<double> vout(sz);
        std::transform(std::execution::par_unseq,
                     v.begin(), v.end(), vout.begin(), pf);

        auto t1 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> fs = t1 - t0;
        fmt::print("Time to apply std::transform"
                  " on {} elements: {}\n", sz, fs.count());
    }
}
```

# C++ Algorithms

**std::transform** is another example of an STL algorithm

With **std::transform**, the lambda function gets the argument by value and returns the result. Why?

Because the output is in a different container

```
whovian@phy504:~/host/CPP$ g++ algo5.cpp -o algo5 -std=c++20 -lfmt -ltbb
Time to apply std::transform on 15000000 elements: 15.260461035
Time to apply std::transform on 15000000 elements: 4.431016585
```

```

#include <random>
#include <algorithm>
#include <fmt/core.h>
#include <chrono>
#include <execution>
#include <cmath>
constexpr int sz = 15'000'000;

int main () {
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, sz);
    std::vector<double> v(sz);
    for (auto& x : v) // by l-value reference
        x = dist(mt);

    auto pf = [] (double n) {
        return std::sqrt(std::sin(n)*std::cos(n));
    };

    auto t0 = std::chrono::high_resolution_clock::now();

    std::vector<double> vout;
    std::transform(v.begin(), v.end(),
                  std::back_inserter(vout), pf);

    auto t1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> fs = t1 - t0;
    fmt::print("Time to apply std::transform"
              " on {0} elements: {1}\n", sz, fs.count());
}

```

# C++ Algorithms

```

whovian@phy504:~/host/CPP$ g++ algo6.cpp -o algo6 -std=c++20 -lfmt -ltbb
whovian@phy504:~/host/CPP$ ./algo6
Time to apply std::transform on 15000000 elements: 19.77442037

```

What if we don't pre-allocate memory on the output vector?

We need **STL  
inserters** (Lec. 3).

Slow! (15.3s vs. 19.8s on a single core)!

With STL inserters, we can't do parallelization (order of exec matters)

# C++ Algorithms

```
#include <random>
#include <algorithm>
#include <fmt/core.h>
#include <chrono>
#include <execution>
#include <cmath>
constexpr int sz = 15'000'000; whovian@phy504:~/host/CPP$ g++ algo7.cpp -o algo7 -std=c++20 -lfmt -ltbb
Time to apply std::transform on 15000000 elements: 0.836872376

int main () {
    std::random_device rd; std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, sz);

    std::vector<double> v1(sz);
    for (auto& x : v1) // by l-value reference
        x = dist(mt);

    std::vector<double> v2(sz);
    for (auto& x : v2) // by l-value reference
        x = dist(mt);

    auto pf = [] (double n1, double n2) { // binary operator
        return n1 + n2;
    };
    {
        auto t0 = std::chrono::high_resolution_clock::now();

        std::vector<double> vout(sz);
        std::transform(std::execution::par_unseq, v1.begin(), v1.end(),
                      v2.begin(), vout.begin(), pf);

        auto t1 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> fs = t1 - t0;
        fmt::print("Time to apply std::transform"
                  " on {} elements: {}\n", sz, fs.count());
    }
}
```

std::transform is also compatible with binary operators (if we want, for instance, to sum two vectors)

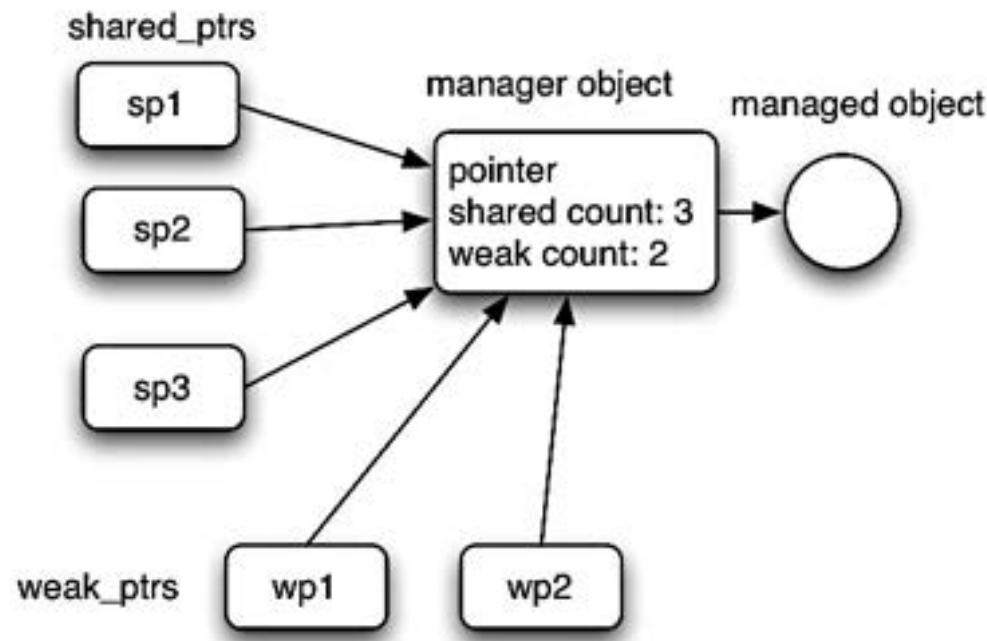
# C++ Algorithms

There are many interesting Algorithms in the STL  
Some STL algorithms can run in parallel execution.  
Some need unary operators (ops), some binary ops

Standard library algorithms for which parallelized versions are provided		
<ul style="list-style-type: none"><li>• <code>std::adjacent_difference</code></li><li>• <code>std::adjacent_find</code></li><li>• <code>std::all_of</code></li><li>• <code>std::any_of</code></li><li>• <code>std::copy</code></li><li>• <code>std::copy_if</code></li><li>• <code>std::copy_n</code></li><li>• <code>std::count</code></li><li>• <code>std::count_if</code></li><li>• <code>std::equal</code></li><li>• <code>std::fill</code></li><li>• <code>std::fill_n</code></li><li>• <code>std::find</code></li><li>• <code>std::find_end</code></li><li>• <code>std::find_first_of</code></li><li>• <code>std::find_if</code></li><li>• <code>std::find_if_not</code></li><li>• <code>std::generate</code></li><li>• <code>std::generate_n</code></li><li>• <code>std::includes</code></li><li>• <code>std::inner_product</code></li><li>• <code>std::inplace_merge</code></li><li>• <code>std::is_heap</code></li></ul>	<ul style="list-style-type: none"><li>• <code>std::is_heap_until</code></li><li>• <code>std::is_partitioned</code></li><li>• <code>std::is_sorted</code></li><li>• <code>std::is_sorted_until</code></li><li>• <code>std::lexicographical_compare</code></li><li>• <code>std::max_element</code></li><li>• <code>std::merge</code></li><li>• <code>std::min_element</code></li><li>• <code>std::minmax_element</code></li><li>• <code>std::mismatch</code></li><li>• <code>std::move</code></li><li>• <code>std::none_of</code></li><li>• <code>std::nth_element</code></li><li>• <code>std::partial_sort</code></li><li>• <code>std::partial_sort_copy</code></li><li>• <code>std::partition</code></li><li>• <code>std::partition_copy</code></li><li>• <code>std::remove</code></li><li>• <code>std::remove_copy</code></li><li>• <code>std::remove_copy_if</code></li><li>• <code>std::remove_if</code></li><li>• <code>std::replace</code></li><li>• <code>std::replace_copy</code></li></ul>	<ul style="list-style-type: none"><li>• <code>std::replace_copy_if</code></li><li>• <code>std::replace_if</code></li><li>• <code>std::reverse</code></li><li>• <code>std::reverse_copy</code></li><li>• <code>std::rotate</code></li><li>• <code>std::rotate_copy</code></li><li>• <code>std::search</code></li><li>• <code>std::search_n</code></li><li>• <code>std::set_difference</code></li><li>• <code>std::set_intersection</code></li><li>• <code>std::set_symmetric_difference</code></li><li>• <code>std::set_union</code></li><li>• <code>std::sort</code></li><li>• <code>std::stable_partition</code></li><li>• <code>std::stable_sort</code></li><li>• <code>std::swap_ranges</code></li><li>• <code>std::transform</code></li><li>• <code>std::uninitialized_copy</code></li><li>• <code>std::uninitialized_copy_n</code></li><li>• <code>std::uninitialized_fill</code></li><li>• <code>std::uninitialized_fill_n</code></li><li>• <code>std::unique</code></li><li>• <code>std::unique_copy</code></li></ul>

# C++ Smart Pointers

When using **C pointers**, we learned that we need to employ **RAII** to make sure memory is deleted



**Smart pointers** are an upgraded version (more powerful and equally convenient) of **RAII** trick.

There are 3 types of **smart pointers**: **unique\_ptr** (unique pointer), **shared\_ptr** (shared pointer), **weak\_ptr** (weak pointer)

# C++ Smart Pointers

```
#include <fmt/core.h>
#include <chrono> // We only covered chrono superficially
#include <thread> // We won't have time to cover threads
#include <memory> // smart pointers
constexpr int sz = 50;
constexpr int sz2 = 30'000'000;

using namespace std::this_thread;      // sleep_for, sleep_until
using namespace std::chrono_literals; // ns, us, ms, s, h, etc.
using std::chrono::system_clock;

int main()
{
    for(int i=0; i<sz; i++)
    {
        std::unique_ptr<int[]> p(new int[sz2]);
        for(int j=0; j<sz2; j++)
        {
            p[j] = j;
        }
        // Here, the std::unique_ptr gets out of scope
        // std::unique_ptr automatically deletes the
        // raw C pointer it holds.

        // How does it know how to delete the raw C pointer?
        // The template argument <int []> indicates the
        // type of C pointer (in this case something it knows
        // how to delete it)

        // No memory leak!!!!!!
        sleep_until(system_clock::now() + 1s);
        fmt::print("\nIteration {0} completed\n", i);
    }
}
```

Unique Pointer  
**(unique\_ptr)**  
uniquely own its  
resource and  
deletes it when the  
object is destroyed

# C++ Smart Pointers

```
#include <fmt/core.h>
#include <chrono> // We only covered chrono superficially
#include <thread> // We won't have time to cover threads
#include <memory> // smart pointers
constexpr int sz = 50;
constexpr int sz2 = 30'000'000;

using namespace std::this_thread; // sleep_for, sleep_until
using namespace std::chrono_literals; // ns, us, ms, s, h, etc.
using std::chrono::system_clock;

int main()
{
    for(int i=0; i<sz; i++)
    {
        std::unique_ptr<int[]> p(new int[sz2]);
        for(int j=0; j<sz2; j++)
        {
            p[j] = j;
        }
        // Only one unique_ptr can hold the data
        // Move is ok (remember how move works!)
        std::unique_ptr<int[]> p2 = std::move(p);

        // here p doesn't hold the data anymore
        if (!p) {
            fmt::print("Confirmation p doesn't hold "
                      "the data anymore\n");
        }
        if (p2) {
            fmt::print("Confirmation p2 holds the data\n");
        }

        // No memory leak!!!!!!
        sleep_until(system_clock::now() + 1s);
        fmt::print("\nIteration {0} completed\n", i);
    }
}
```

Why **unique\_ptr** is called unique pointer?

Because only one **unique\_ptr** can hold the data (so no copy allowed! Move is ok!)

```
#include <fmt/core.h>
#include <chrono> // We only covered chrono superficially
#include <thread> // We won't have time to cover threads
#include <memory> // smart pointers
constexpr int sz = 50;
constexpr int sz2 = 30'000'000;

using namespace std::this_thread;      // sleep_for, sleep_until
using namespace std::chrono_literals; // ns, us, ms, s, h, etc.
using std::chrono::system_clock;

int main()
{
    for(int i=0; i<sz; i++)
    {
        std::shared_ptr<int[]> p(new int[sz2]);
        for(int j=0; j<sz2; j++)
        {
            p[j] = j;
        }

        // multiple shared_ptr can hold the same data
        std::shared_ptr<int[]> p2 = p;

        if (p) {
            fmt::print("Confirmation p holds the data\n");
        }
        if (p2) {
            fmt::print("Confirmation p2 holds the data\n");
        }

        // No memory leak!!!!!!
        sleep_until(system_clock::now() + 1s);
        fmt::print("\nIteration {0} completed\n", i);
    }
}
```

# C++ Smart Pointers

If you want multiple pointers to hold the same data: use  
**shared\_ptr**

**shared\_ptr**  
contains an internal counter and only deletes if the counter -> 0

# C++ Smart Pointers

The elephant in the room. Why using smart pointers instead of C++ containers (ex: vectors)?

Best to use C++ containers (ex: vectors) 99% of the time

Is there any use for unique\_ptr with array?

Asked 9 years, 11 months ago Modified 7 months ago Viewed 234k times

 std::unique\_ptr has support for arrays, for instance:

304  std::unique\_ptr<int[]> p(new int[10]);

 but is it needed? probably it is more convenient to use std::vector or std::array .

 Do you find any use for that construct?

[c++](#) [c++11](#) [smart-pointers](#) [unique\\_ptr](#)

# C++ Smart Pointers

```
#include <fmt/core.h>
#include <memory> // smart pointers
using std::unique_ptr;
typedef gsl_integration_workspace gsl_int_work; // nickname

// This is a template Class (C++ metaprogramming)
// We will overview template next class (Lecture 14)
template<typename T> class Deleter;

// Template specialization. This class will work as a functor
template<> class Deleter<gsl_int_work> {
public:
    void operator()(gsl_int_work* ptr) {
        gsl_integration_workspace_free(ptr);
        return;
    }
};

// unique_ptr<A,B> has two template arguments
// A = type of the raw C pointer it holds
// B = functor that teaches the smart pointer how
//     to delete the raw C pointer
typedef unique_ptr<gsl_int_work, Deleter<gsl_int_work>>
    cpp_gsl_integration_workspace;
```

## Smart Pointers and the GSL library

Can smart  
pointers wrap C  
pointers with  
complicated API  
for deletion?

YES, with a few  
caveats (and help  
from **Templates**)

```
#include <gsl/gsl_integration.h>
#include <fmt/core.h>
#include <memory> // smart pointers
using std::unique_ptr;
typedef gsl_integration_workspace gsl_int_work; // nickname

template<typename T> class Deleter;
template<> class Deleter<gsl_int_work> {
public:
    void operator()(gsl_int_work* ptr) {
        gsl_integration_workspace_free(ptr);
        return;
    }
};
typedef unique_ptr<gsl_int_work, Deleter<gsl_int_work>>
    cpp_gsl_integration_workspace;

double f (double x, void* params) {
    double alpha = *(double *) params;
    return log(alpha*x) / sqrt(x);
}

int main (void) {
    // unique_ptr constructor can have as an arg the raw pointer
    cpp_gsl_integration_workspace
        w(gsl_integration_workspace_alloc (1000));

    double result, error, alpha = 1.0;
    gsl_function F; F.function = &f; F.params = &alpha;
    const double int_min = 0.0, int_max = 1.0;
    const double rel_error = 1e-7, abs_error = 0.0;

    // .get() returns the raw C pointer. That defeats
    // the encapsulation of unique_ptr (no copy allowed)
    // But it works, as long as you make sure that w is in scope
    gsl_integration_qags (&F, int_min, int_max, abs_error,
        rel_error, 1000, w.get(), &result, &error);

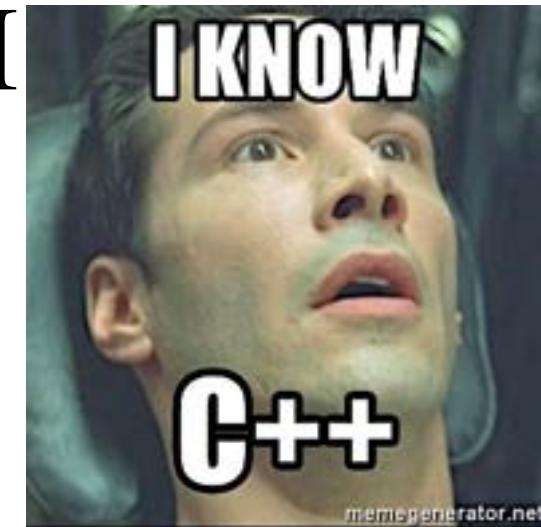
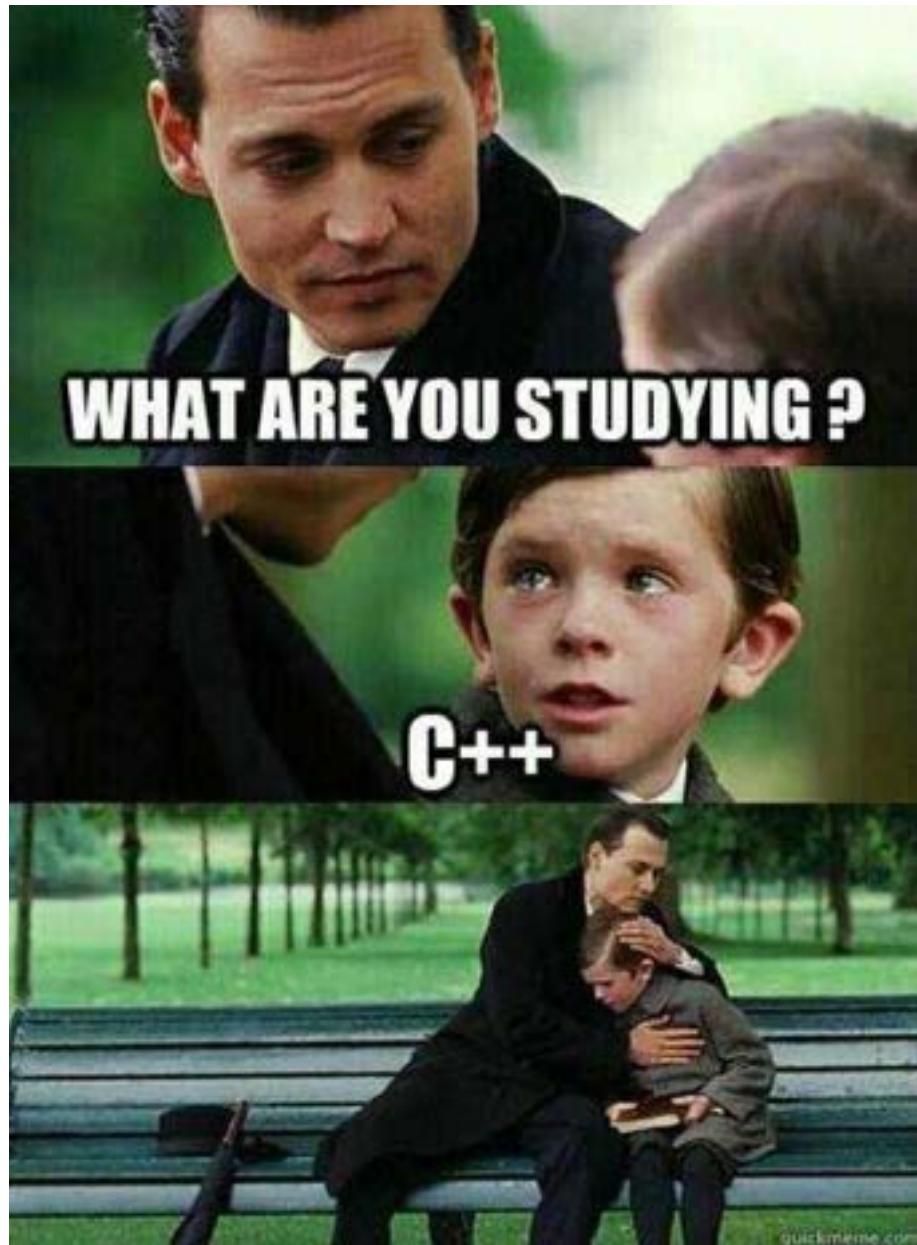
    fmt::print ("result      = {}\n", result); whovian@phy504:~/host/CPP$ g++ sm5.cpp -o sm5 -std=c++20 -lfmt -lgsl -lgslcblas
    fmt::print ("exact result = {}\n", -4.0);   whovian@phy504:~/host/CPP$ ./sm5
    result      = -4.0000000000000085
    exact result = -4
}
```

# C++ Smart Pointers

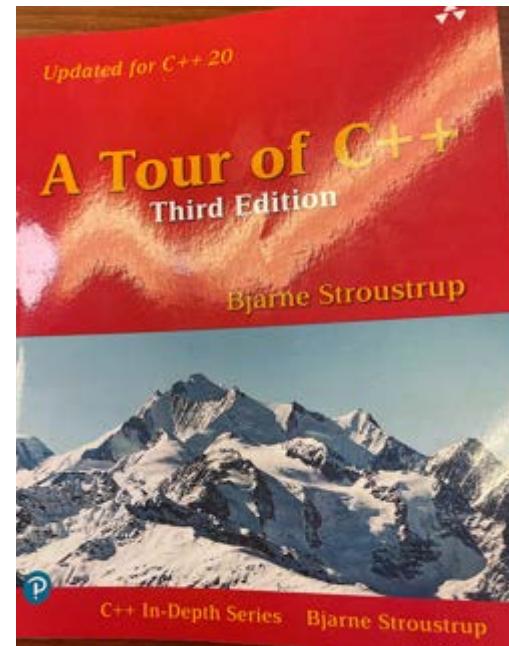
Can smart pointers wrap  
C pointers with  
complicated API for  
deletion?

YES, with a few caveats  
(and help from  
**Templates**)

# Lecture 37: C++ part XIII



## Suggested Literature



```

#include <algorithm>
#include <fmt/core.h>
#include <vector>

int main ()
{
    std::vector<int> lt = {7, 5, 16, 8, 10};
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    lt.push_back(13); // add to the back of the list
    // There is no push_front on vectors
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    // find elements in the list with the STL algorithm
    auto it = std::find(lt.begin(), lt.end(), 16);
    if (it != lt.end())
        lt.insert(it, 42); // inserts value before pos

    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");
    return 0;
}

```

# Linear Algebra in C++

Memory in vectors  
are **contiguous**

There is no  
**push\_front**

(vectors are not a  
double linked list)

```

whovian@phy504:~/host/CPP$ g++ vector.cpp -o vector1 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./vector1
7, 5, 16, 8, 10,
7, 5, 16, 8, 10, 13,
7, 5, 42, 16, 8, 10, 13,

```

```
#include <algorithm>
#include <fmt/core.h>
#include <vector>

int main ()
{
    std::vector<int> lt = {7, 5, 16, 8, 10};

    fmt::print("First element {0}\n", lt[0]);
    fmt::print("First element {0}\n", lt.at(0));

    const auto sz = lt.size();
    fmt::print("Size of the vector: {0}\n", sz);

    // This is an interesting concept (capacity)
    // vector sometimes holds more memory than its size
    const auto cap = lt.capacity();
    fmt::print("Capacity of the vector: {0}\n", cap);

    // Why? to allow fast push back
    lt.push_back(100);

    const auto sz2 = lt.size();
    fmt::print("Size of the vector: {0}\n", sz2);

    const auto cap2 = lt.capacity();
    fmt::print("Capacity of the vector: {0}\n", cap2);

    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    return 0;
}
```

# Linear Algebra in C++

Memory in vectors is **contiguous**

So totally ok to use **operator[]** or **.at()**  
(safer version – checks if out of bound)

```
whovian@phy504:~/host/CPP$ g++ vector2.cpp -o vector2 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./vector2
First element 7 ←
First element 7 ←
Size of the vector: 5
Capacity of the vector: 5
Size of the vector: 6
Capacity of the vector: 10
7, 5, 16, 8, 10, 100,
```

```
#include <algorithm>
#include <fmt/core.h>
#include <vector>

int main ()
{
    std::vector<int> lt = {7, 5, 16, 8, 10};

    fmt::print("First element {0}\n", lt[0]);
    fmt::print("First element {0}\n", lt.at(0));

    const auto sz = lt.size();
    fmt::print("Size of the vector: {0}\n", sz);

    // This is an interesting concept (capacity)
    // vector sometimes holds more memory than its size
    const auto cap = lt.capacity();
    fmt::print("Capacity of the vector: {0}\n", cap);

    // Why? to allow fast push back
    lt.push_back(100);

    const auto sz2 = lt.size();
    fmt::print("Size of the vector: {0}\n", sz2);

    const auto cap2 = lt.capacity();
    fmt::print("Capacity of the vector: {0}\n", cap2);

    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    return 0;
}
```

# Linear Algebra in C++

Capacity is usually larger than size (vector holds more memory than currently needed)

Why? To allow fast future expansion via **push\_back**

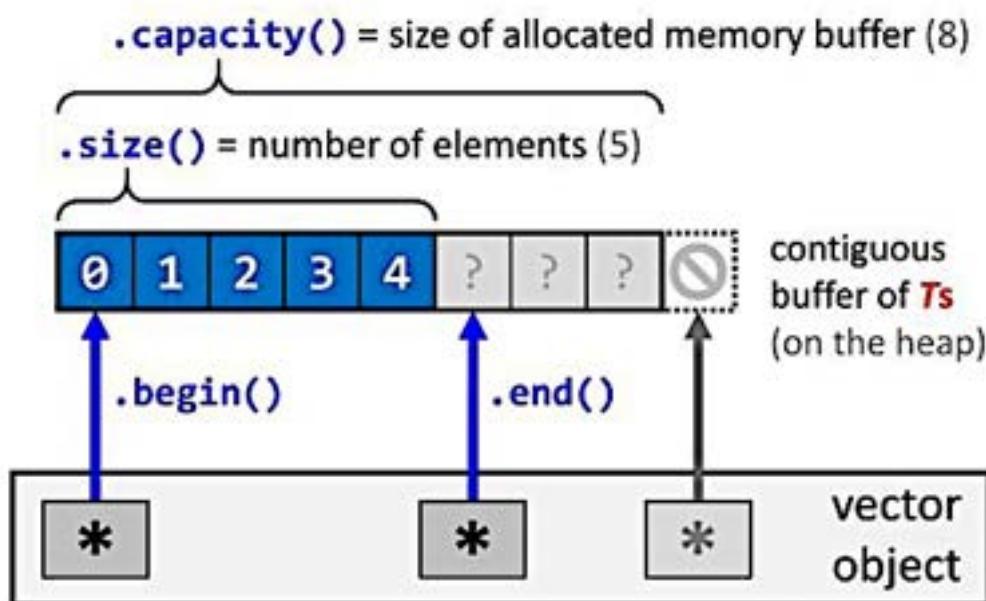
```
whovian@phy504:~/host/CPP$ g++ vector2.cpp -o vector2 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./vector2
First element 7
First element 7
Size of the vector: 5
Capacity of the vector: 5 ←
Size of the vector: 6
Capacity of the vector: 10 ←
7, 5, 16, 8, 10, 100,
```

# Linear Algebra in C++

Capacity is usually larger than size

**push\_back** is cheap as long as its size < capacity

`std::vector<T>` Size vs. Capacity



What happens when calling **push\_back** with **sz == capacity**?

Slower: the entire vector needs to move to a new location with extra capacity

```
#include <algorithm>
#include <fmt/core.h>
#include <vector>
#include <chrono>
constexpr int bf = 250'000'000;

int main ()
{
    { // just creating local stack
        std::vector<int> lt;
        // change the capacity without changing its size
        lt.reserve(bf+1);

        auto t0 = std::chrono::high_resolution_clock::now();

        for (int i=0; i<bf; i++)
            lt.push_back(i);

        auto t1 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> fs = t1 - t0;

        fmt::print("Time to executie the loop (capacity\n"
                  "    pre-allocated): {0}\n", fs.count());
    }
    {
        std::vector<int> lt;

        auto t0 = std::chrono::high_resolution_clock::now();

        for (int i=0; i<bf; i++)
            lt.push_back(i);

        auto t1 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> fs = t1 - t0;

        fmt::print("Time to executie the loop (capacity NOT\n"
                  "    pre-allocated): {0} sec\n", fs.count());
    }
    return 0;
}
```

# Linear Algebra in C++

What happens when calling  
**push\_back**  
with **sz == capacity**?

Super slow: the entire  
vector needs to move to a  
new location with extra  
capacity

```
whovian@phy504:~/host/CPP$ nano vector3.cpp
whovian@phy504:~/host/CPP$ g++ vector3.cpp -o vector3 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./vector3
Time to executie the loop (capacity pre-allocated): 23.943483455
Time to executie the loop (capacity NOT pre-allocated): 25.488765179 sec
```



```

#include <fmt/core.h>
#include <vector>
#include <mdspan> // from external library on Docker
namespace stdex = std::experimental;

int main ()
{
    std::vector<int> lt;
    lt.resize(9); // changing vector size
                  // (capacity will readjust accordingly)

    // can we this vector as 9 x 9
    auto ms = stdex::mdspan(lt.data(), 3, 3);

    for(size_t i=0; i != ms.extent(0); i++) {
        for(size_t j=0; j != ms.extent(1); j++) {
            ms[i, j] = i + j;
        }
    }

    for(size_t i=0; i != ms.extent(0); i++) {
        for(size_t j=0; j != ms.extent(1); j++) {
            fmt::print("a({0},{1}) = {2}\n", i, j, ms[i, j]);
        }
    }

    return 0;
}

```

# Linear Algebra in C++

Can we stride an 1D  
vector to be a matrix  
(or a tensor)?

In C++23, we can!

Not yet implemented  
even on gcc13 (so we  
use an external library)

```

whovian@phy504:~/host/CPP$ g++ vector4.cpp -o vector4 -std=c++23 -lfmt
whovian@phy504:~/host/CPP$ ./vector4
a(0,0) = 0
a(0,1) = 1
a(0,2) = 2
a(1,0) = 1
a(1,1) = 2
a(1,2) = 3
a(2,0) = 2
a(2,1) = 3
a(2,2) = 4

```

# Linear Algebra in C++

C++ STL doesn't have high performance library for linear algebra. Much better solution: [Armadillo lib](#)



**Armadillo**  
C++ library for linear algebra & scientific computing

[About](#) [Documentation](#) [Questions](#) [Speed](#) [Contact](#) [Download](#)

- Armadillo is a high quality linear algebra library (matrix maths) for the C++ language, aiming towards a good balance between speed and ease of use
- Provides high-level syntax and functionality deliberately similar to Matlab
- Useful for algorithm development directly in C++, or quick conversion of research code into production environments
- Provides efficient classes for vectors, matrices and cubes; dense and sparse matrices are supported
- Integer, floating point and complex numbers are supported
- A sophisticated expression evaluator (based on template meta-programming) automatically combines several operations to increase speed and efficiency
- Dynamic evaluation automatically chooses optimal code paths based on detected matrix structures
- Various matrix decompositions (eigen, SVD, QR, etc) are provided through integration with LAPACK, or one of its high performance drop-in replacements (eg. MKL or OpenBLAS)
- Can automatically use OpenMP multi-threading (parallelisation) to speed up computationally expensive operations
- Distributed under the permissive Apache 2.0 license, useful for both open-source and proprietary (closed-source) software
- Can be used for machine learning, pattern recognition, computer vision, signal processing, bioinformatics, statistics, finance, etc
- [download latest version](#) | [git repo](#) | [browse documentation](#)

---

Supported by:



```
#include <iostream>
#include <armadillo>

using namespace std;

int main()
{
    // Matrix 5 x 5, fill with random numbers
    arma::mat A(5, 5, arma::fill::randn);
    arma::mat B(5, 5, arma::fill::randn);

    arma::mat C = A * B;
    cout << C << endl;

    arma::vec d = C.col(0); // first column of C matrix
    cout << d << endl;

    // Two ways of computing the inverse matrix
    arma::mat D = C.i();
    cout << C * D << endl;

    arma::mat E = inv(C, arma::inv_opts::no_ugly);
    cout << C * E << endl;

    // Computing Eigenvalues
    arma::mat F = C.t()*C; // generate symmetric matrix
    arma::vec eigval = arma::eig_sym( F );
    cout << eigval << endl;

    return 0;
}
```

# Linear Algebra in C++

C++ STL doesn't have high performance library for linear algebra.

Much better solution:

**Armadillo lib**

# Linear Algebra in C++

C++ STL doesn't have high performance library for linear algebra. Much better solution: **Armadillo lib**

```
whovian@phy504:~/host/CPP$ g++ vector5.cpp -o vector5 -std=c++17 -DARMA_DONT_USE_WRAPPER -larmadillo -lblas -llapack
whovian@phy504:~/host/CPP$ ./vector5
  0.3983  -1.0662  -1.5257  -3.1505  -1.1145
 -2.2801  -0.2953  -1.3318   0.0308  -0.2634
  4.4399   1.2315  -0.1849   0.9604  -1.2402
  1.6139   1.3376   1.8784   1.9148   0.2361
  1.6675  -1.7002  -2.2438  -4.2205  -0.5873

  0.3983
 -2.2801
  4.4399
  1.6139
  1.6675

  1.0000e+00   2.2204e-16  -3.3307e-16   4.4409e-16   8.8818e-16
 -1.6653e-16   1.0000e+00  -2.2204e-16   1.3878e-15   7.7716e-16
  4.4409e-16  -1.7764e-15   1.0000e+00  -2.2204e-15  -1.7764e-15
  2.7756e-16  -4.9960e-16   1.5266e-16   1.0000e+00  -7.7716e-16
    0     8.8818e-16  -2.2204e-16  -8.8818e-16   1.0000e+00

  1.0000e+00   2.2204e-16  -3.3307e-16   4.4409e-16   8.8818e-16
 -1.6653e-16   1.0000e+00  -2.2204e-16   1.3878e-15   7.7716e-16
  4.4409e-16  -1.7764e-15   1.0000e+00  -2.2204e-15  -1.7764e-15
  2.7756e-16  -4.9960e-16   1.5266e-16   1.0000e+00  -7.7716e-16
    0     8.8818e-16  -2.2204e-16  -8.8818e-16   1.0000e+00

  0.0185
  0.4956
  3.0108
 32.5368
 50.0915
```

Covers C++11, C++14, and C++17



# C++ Templates

The Complete Guide

SECOND EDITION

David VANDEVOORDE  
Nicolai M. JOSUTTIS  
Douglas GREGOR



## Templates

C++ template metaprogramming is a large topic that we will only glance

Advanced feature

Takes time to study / debug metaprogramming

Suggested Literature

Covers C++11, C++14, and C++17



# C++ Templates

The Complete Guide

SECOND EDITION

David VANDEVOORDE  
Nicolai M. JOSUTTIS  
Douglas GREGOR



# Templates

When writing a program with multiple files:

**Templates must always be on headers**

Why? Because templates are metaprograms (metafunctions, metaclasses...)

```
#include <iostream>
#include <string>

using std::string;
using std::cout;
using std::endl;

// We write one single metafunction
// The compiler creates appropriate
// functions for every type
template<typename T>
T mymax(T a, T b)
{
    return b < a ? a : b;
}

int main()
{
    cout << "My first program with templates" << endl;
    cout << mymax<double>(2.0, 3.0) << endl;
    cout << mymax<int>(2, 3) << endl;
    std::string s1 = "mathematics";
    std::string s2 = "math";
    cout << mymax<string>(s1, s2) << endl;
}
```

Templates args: <arg1, arg2,...>



```
whovian@phy504:~/host/CPP$ g++ template1.cpp -o template1 -std=c++20
whovian@phy504:~/host/CPP$ ./template1
My first program with templates
3
3
mathematics
```

# Templates

Template arguments are usually types  
(can be integers)

Templates are converted to normal functions during compilation

```
#include <iostream>
#include <string>

using std::string;
using std::cout;
using std::endl;

template<typename T>
T mymax(T a, T b)
{
    return b < a ? a : b;
}

int main()
{
    cout << "My second program with templates" << endl;

    // The second arg being an int is ok. Why?
    // We explicitly made the template arg = double
    // So the argument will be cast to double
    cout << mymax<double>(2.0, 3) << endl;

    // Same here: the second arg will be cast to int
    cout << mymax<int>(2, 3.0) << endl;

    std::string s2 = "math";
    // Here: String literals are const char*, but
    // an unnamed string will be created (and std::string
    // accept string literals in its constructor)
    cout << mymax<string>("mathematics", s2) << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ template2.cpp -o template2 -std=c++20
whovian@phy504:~/host/CPP$ ./template2
My second program with templates
3
3
mathematics
```

# Templates

Template arguments can be deduced from arg's types

We need to be a bit careful here. No casting allowed if template args are deduced

```
#include <iostream>
#include <string>
using std::string; using std::cout; using std::endl;

template<typename T>
T mymax(T a, T b) {
    return b < a ? a : b;
}

int main() {
    cout << "My third program with templates" << endl;
    // compiler error!
    cout << mymax(2, 3.5) << endl;
    // compiler error!
    string s2 = "math";
    cout << mymax("mathematics", s2) << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ template3.cpp -o template3 -std=c++20
template3.cpp: In function 'int main()':
template3.cpp:19:16: error: no matching function for call to 'mymax(int, double)'
  19 |     cout << mymax(2, 3.5) << endl;
               |
template3.cpp:9:3: note: candidate: 'template<class T> T mymax(T, T)'
  9 | T mymax(T a, T b)
               |
template3.cpp:9:3: note:   template argument deduction/substitution failed:
template3.cpp:19:16: note:   deduced conflicting types for parameter 'T' ('int' and 'double')
  19 |     cout << mymax(2, 3.5) << endl;
               |
template3.cpp:23:16: error: no matching function for call to 'mymax(const char [12], std::string&)'
  23 |     cout << mymax("mathematics", s2) << endl;
               |
template3.cpp:9:3: note: candidate: 'template<class T> T mymax(T, T)'
  9 | T mymax(T a, T b)
               |
template3.cpp:9:3: note:   template argument deduction/substitution failed:
template3.cpp:23:16: note:   deduced conflicting types for parameter 'T' ('const char*' and 'std::__cxx11::basic_string<char>')
  23 |     cout << mymax("mathematics", s2) << endl;
               |
```

# Templates

We need to be a bit careful here. No casting allowed if template args are deduced

```

#include <iostream>
#include <string>

using std::string;
using std::cout;
using std::endl;

// Here the type of the first argument
// specifies the type of the return
template<typename T1, typename T2>
T1 mymax(T1 a, T2 b)
{
    return b < a ? a : b;
}

int main()
{
    cout << "My Fourth program with templates" << endl;

    // Not a compiler error anymore, but...
    // the following two lines produce different answers!
    cout << mymax(2, 3.5) << endl;  Solving this issue will
    cout << mymax(3.5, 2) << endl;      take a few slides
}

```

```

whovian@phy504:~/host/CPP$ g++ template4.cpp -o template4 -std=c++20
whovian@phy504:~/host/CPP$ ./template4
My Fourth program with templates
3
3.5

```

# Templates

We can have multiple **template arguments**  
 (even infinite w/  
**variadic templates**)

Here, we still need  
 to be a bit careful  
 with arg. deduction

```
#include <iostream>
#include <string>
using std::string; using std::cout; using std::endl;

// Here, we did something different. We made the
// return type independent of the types of the
// function arguments. Therefore, we will always
// need to write the return type explicitly as a
// template argument when calling the function.

// We strategically put the return type as the first
// template argument, so T2 and T3 can be deduced
// from the types of the function arguments
// (partially template argument deduction)
template<typename T1, typename T2, typename T3>
T1 mymax(T2 a, T3 b)
{
    return b < a ? a : b;
}

int main()
{
    cout << "My fifth program with templates" << endl;

    // We explicitly included the return type
    // as a template argument. Second and third
    // template arguments will be deduced from
    // the types of the function arguments.
    cout << mymax<double>(2, 3.5) << endl;
    cout << mymax<double>(3.5, 2) << endl;
}
```

# Templates

We can have multiple  
**template arguments.**

We still need to be a bit  
careful with arg.  
deduction

First solution: (a bit  
ugly): independent  
return type

```
whovian@phy504:~/host/CPP$ g++ template5.cpp -std=c++20
whovian@phy504:~/host/CPP$ ./template5
My fifth program with templates
3.5
3.5
```

```

#include <iostream>
#include <string>
#include <type_traits>
using std::string; using std::cout; using std::endl;

template<typename T1, typename T2>
std::common_type_t<T1,T2> mymax(T1 a, T2 b)
{
    return b < a ? a : b;
}

int main()
{
    cout << "My sixth program with templates" << endl;

    int x = 2;
    double y = 5.1;
    cout << std::fixed << mymax(x, y) << endl;
    cout << std::fixed << mymax(y, x) << endl;

    int z1 = 5;
    double z2 = 2.2;
    cout << std::fixed << mymax(z1, z2) << endl;
    cout << std::fixed << mymax(z2, z1) << endl;

    return 0;
}

```

# Templates

STL **type\_traits** rescue us

**std::common\_type\_t** determines the common type among all types, that is the type all Ts can be implicitly converted to. If such a type exists, the member type names that type. Otherwise, there is no member type (compiler error)

```

whovian@phy504:~/host/CPP$ g++ template6.cpp -o template6 -std=c++20
whovian@phy504:~/host/CPP$ ./template6
My sixth program with templates
5.100000
5.100000
5.000000
5.000000

```

```

#include <iostream>
#include <string>
#include <type_traits>
using std::string; using std::cout; using std::endl;

template<typename T1, typename T2>
std::common_type_t<T1,T2> mymax(T1 a, T2 b)
{
    return b < a ? a : b;
}

int main() {
    int x = 2;
    std::string y = "Testing";
    cout << std::fixed << mymax(x, y) << endl;
}

```

```

whovian@phy504:~/host/CPP$ g++ template7.cpp -o template7 -std=c++20
template7.cpp: In function 'int main()':
template7.cpp:18:30: error: no matching function for call to 'mymax(int&, std::string&)'
    18 |     cout << std::fixed << mymax(x, y) << endl;
                  |
template7.cpp:7:27: note: candidate: 'template<class T1, class T2> std::common_type_t<_Tp1, _Tp2> mymax(T1
, T2)'
    7 |     std::common_type_t<T1,T2> mymax(T1 a, T2 b)
                  |
template7.cpp:7:27: note:   template argument deduction/substitution failed:
In file included from /usr/include/c++/12/bits/move.h:57,
                 from /usr/include/c++/12/bits/exception_ptr.h:43,
                 from /usr/include/c++/12/exception:168,
                 from /usr/include/c++/12/ios:39,
                 from /usr/include/c++/12/ostream:38,
                 from /usr/include/c++/12/iostream:39,
                 from template7.cpp:1:
/usr/include/c++/12/type_traits: In substitution of 'template<class ... _Tp> using common_type_t = typename std::common_type::type [with _Tp = {int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>}]]':
template7.cpp:7:27:   required by substitution of 'template<class T1, class T2> std::common_type_t<_Tp1, _Tp2> mymax(T1, T2) [with T1 = int; T2 = std::__cxx11::basic_string<char>]'
template7.cpp:18:30:   required from here
/usr/include/c++/12/type_traits:2622:11: error: no type named 'type' in 'struct std::common_type<int, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>>'
2622 |     using common_type_t = typename common_type<_Tp...>::type;
                  |

```

# Templates

## std::common\_type\_t

Example: no common type between **string** and **int**.

```
#include <iostream>
#include <string>
#include <type_traits>
using std::string; using std::cout; using std::endl;

template<typename T1, typename T2>
std::common_type_t<T1,T2> mymax(T1 a, T2 b) {
    return b < a ? a : b;
}

// overloading for pointers of the same type
template<typename T>
T* mymax(T* a, T* b) {
    cout << "Template overloading (pointer)" << endl;
    return *b < *a ? a : b;
}

int main() {
    int x = 2;
    int y = 5;
    cout << std::fixed << mymax(x, y) << endl;

    int* z1 = &x;
    int* z2 = &y;
    auto z3 = mymax(z1, z2);
    cout << std::fixed << *z3 << endl;
}
```

Templates  
Function  
overloading also  
works with  
template  
functions

Ex: expand  
template function  
to work with  
pointers

```
whovian@phy504:~/host/CPP$ g++ template8.cpp -o template8 -std=c++20
whovian@phy504:~/host/CPP$ ./template8
5
Template overloading (pointer)
5
```